Universidad de Alcalá
Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

Trabajo Fin de Máster

DETECCIÓN Y SEGUIMIENTO DE OBJETOS MEDIANTE PRECISIÓN-TRACKING

Autor: Samuel Pardo Alía

Tutor/es: Elena López Guillén

2018

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

Trabajo Fin de Máster

“Detección y seguimiento de objetos mediante precisión-tracking”

Samuel Pardo Alía

2018

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

Trabajo Fin de Máster

"Detección y seguimiento de objetos mediante precisión-tracking"

Autor: Samuel Pardo Alía

Tutor/es: Elena López Guillén

TRIBUNAL:

Presidente: Dña. Marta Marrón Romera

Vocal 1: D. Ignacio Parra Alonso

Vocal 2: Dña. Elena López Guillén

FECHA:.....

RESUMEN

El trabajo presente en este libro consiste en la implementación de un sistema de detección y seguimiento de objetos preciso y en tiempo real de una escena de tráfico urbano común empleando las técnicas de precision-tracking descritas en [1] sobre la plataforma de desarrollo de software robótico ROS.

La finalidad del sistema implementado es poder identificar en todo momento la posición y velocidad de los objetos que rodean a un vehículo propio que se encuentra provisto de un LiDAR 3D y una cámara, entre otros sensores, que aportan la información del entorno.

Por último, se realizará una comparativa en la precisión en la estimación de velocidad de los objetos contrastando los resultados obtenidos por un filtro de Kalman con el algoritmo de precisión-tracking.

PALABRAS CLAVE: Detección, Seguimiento, ROS, Kalman, Precision-Tracking.

ABSTRACT

The work described in this book consists of an accurate real-time object detection and tracking on a common urban traffic scene using the precision-tracking techniques [1] by the robotic software framework ROS.

The system's main purpose is to be able to identify at any moment the objects' position and velocity that surround an ego-vehicle provided with a 3D LiDAR and a camera, among other sensors, that supply the information of the background.

Finally, it will be made a precision's comparison with the objects' velocity estimation matching the results obtained by a Kalman filter with the precision-tracking algorithm.

KEYWORDS: Detection, Tracking, ROS, Kalman, Precision-Tracking.

RESUMEN EXTENDIDO

Uno de los aspectos más críticos a la hora de desarrollar un vehículo autónomo consiste en poder percibir e interpretar el entorno que le rodea para poder llevar a cabo las acciones necesarias para garantizar la seguridad tanto del propio vehículo y los pasajeros que lo ocupan como de otros vehículos, personas y la infraestructura urbana circundante.

En esta tarea de percepción juega un papel fundamental la detección y seguimiento de los objetos, que permite al vehículo poder identificar la posición actual y las pasadas, y en base a la trayectoria descrita, poder estimar la posición futura y así planificar sus acciones de movimiento de forma dinámica.

Aunque ha habido un gran progreso en la detección y seguimiento de objetos, sigue siendo un problema de interés en la investigación debido a las restricciones producidas por oclusiones, cambios en el punto de vista, o iluminación, que existen en un entorno real. Por lo tanto, existe una demanda en el desarrollo de un enfoque robusto que incorpore tanto el proceso de detección como seguimiento en un mismo marco.

En el caso de este trabajo, la información del entorno será adquirida por un láser LiDAR 3D de 16 haces y por una cámara estéreo. El LiDAR 3D es un sensor que proporciona información espacial de gran alcance y en tiempo real alrededor del vehículo, mientras que la cámara se emplea para reforzar las medidas de distancia aportando el color a cada uno de los puntos reflejados. La forma de combinar la información de distancia y de color es realizando una fusión sensorial que permita calibrar ambos sensores respecto a un marco de referencia común.

Los datos procedentes de los sensores se obtienen en forma de nube de puntos empleando la librería PCL, y son tomados por el sistema como entrada para realizar la detección de los múltiples objetos de interés que puedan encontrarse en la escena, de los que se va a realizar el seguimiento tanto de su posición como de su dimensión. El seguimiento se basa en un filtro bayesiano de Kalman para abordar tanto la asociación de datos como estimar el modelo de movimiento del objeto seguido.

Para evitar los problemas de precisión y robustez que suelen tener en general los algoritmos de seguimiento, se decide implementar las técnicas de precision-tracking que permiten combinar tanto la forma de los objetos como su color garantizando un seguimiento en tiempo real. Este algoritmo se basa en gestionar los esfuerzos computacionales realizando inicialmente una aproximación burda de la distribución de probabilidad de ocupación del objeto en una rejilla, para después refinar la búsqueda, aumentando la precisión con cada iteración que el objeto es detectado.

Finalmente, para evaluar el rendimiento en entornos urbanas reales, se implementan varias escenas en el simulador VREP y una escena con datos reales, y se comparan los resultados obtenidos en cuanto a estimación de la velocidad tanto del filtro de Kalman como de las técnicas de precision-tracking implementadas.

CONTENIDO GENERAL

LISTADO DE FIGURAS

LISTADO DE TABLAS

MEMORIA

Capítulo I

INTRODUCCIÓN

1.1. Presentación

A pesar de ser una idea que ha tomado fuerza en las últimas décadas, las primeras semillas de la conducción autónoma fueron plantadas al poco tiempo del nacimiento del automóvil. Ya en 1925, el ingeniero eléctrico Francis Houdina propuso un automóvil teleoperado por radio, pero no fue hasta 1969, cuando John McCarthy, considerado uno de los padres de la inteligencia artificial, en uno de sus ensayos, describió un vehículo autónomo que podrían controlar los usuarios, que, a pesar de no materializarse, resultó ser el texto de referencia para investigadores en años posteriores.

En la década de los ‘90 se comenzaron a hacer las primeras pruebas reales de conducción autónoma. En la Universidad Carnegie Mellon (CMU), en Pensilvania (EE.UU.) se empieza a estudiar la aplicación de redes neuronales para la identificación de las imágenes que captaba el coche y así poder controlarlo. En 1995, Pomerleau y Jochem, dos investigadores de la CMU, ponen en práctica su sistema de navegación autónoma Navlab, que había nacido 10 años antes como parte de DARPA (Defense Advanced Research Projects Agency) con el objetivo de impulsar la inteligencia artificial. Consiguieron recorrer EE.UU. de costa a costa, desde Pensilvania hasta San Diego, en California.

Esta misma Agencia, en 2002, crea un desafío entre las principales instituciones de investigación ofreciendo un premio de un millón de dólares. A pesar de que en su primera edición en 2004 ninguno de sus participantes fue capaces de completarlo, para muchos esta competición supone un punto de inflexión que lleva a la proliferación de un número creciente de investigaciones en el campo de la conducción autónoma que nos lleva hasta la actualidad.

FIGURA 1.1-1: Imagen de uno de los vencedores de la competición de DARPA.

A pesar de que los vehículos autónomos aún no han desplazado a los vehículos convencionales, desde inicios del siglo XXI hemos visto cómo se van introduciendo distintos sistemas inteligentes, y en los últimos años las principales compañías automovilísticas (General Motors, Mercedes Benz o BMW) han lanzado al mercado sus propias tecnologías de conducción autónoma.

1.2. Contexto

De la realidad prometedora de los vehículos autónomos se hacen eco los principales centros de investigación de todo el mundo. La propia Universidad de Alcalá cuenta con diversos grupos de investigación sobre este tema, siendo uno de los principales el grupo RobeSafe, sin el cual no hubiera sido posible el desarrollo del presente trabajo.

Este grupo, que se centra en los sistemas de percepción aplicados a los campos de la Robótica de Servicios y Seguridad Electrónica, lleva unos años inmerso en el proyecto *Smart Elderly Car* [2] cuya misión se basa en el desarrollo de un automóvil eléctrico autónomo para ayudar a la población de avanzada edad en entornos urbanos.

Este proyecto no sólo se centra en el impacto social de la inclusión del vehículo autónomo en el panorama actual, sino que tiene como objetivo mejorar la seguridad del tráfico para reducir la mortalidad en carretera, además de resultar ser un transporte sostenible para el medioambiente gracias a su motor eléctrico.

Una de las tareas más importantes a la hora de desarrollar un vehículo autónomo es poder percibir el entorno que le rodea para poder actuar en consecuencia y anticiparse a posibles peligros derivados. Tomando esto como referencia el trabajo que se presenta se centra en la implementación de un sistema de seguimiento de la posición y velocidad de los objetos que componen una escena de tráfico urbano habitual.

1.3. Estado del arte

El seguimiento de objetos en una escena tiene importancia en diferentes campos como la seguridad y vigilancia, o el control de tráfico. Su principal objetivo consiste en asociar objetos que se encuentran en instantes de tiempo consecutivos, lo que puede resultar complejo si se desplazan a una gran velocidad o si realizan continuos cambios de trayectoria.

En general el estudio del seguimiento de objetos suele presentar etapas comunes: segmentación y modelado de objetos, localización del objeto en cada *frame* y predicción de su posterior ubicación. Para resolver este problema en los últimos años se han ido tomando distintos enfoques como los que se van a exponer a continuación.

1.3.1. Enfoque de seguimiento visual (VTO)

Uno de los enfoques más empleados se basa en el seguimiento visual, conocido como *Visual Object Tracking* (VOT). El VTO se centra únicamente en la información obtenida desde una cámara, aprovechando las ventajas de la tecnología de video, como su portabilidad y bajo coste; y la madurez de algoritmos de visión artificial.

Se han seguido principalmente 2 tendencias basadas en el enfoque de VTO [3]:

- Asociación de datos: consiste en determinar qué objeto de la escena se corresponde con el mismo objeto en el instante anterior. Es uno de los principales desafíos para realizar un seguimiento de múltiples objetos. Se han hecho esfuerzos orientados hacia modelos de movimiento empleando algoritmos con filtros bayesianos.

- Afinidad y apariencia: los esfuerzos más actuales se centran en características robustas de la apariencia, lo que aumenta el rendimiento y la versatilidad en escenarios más complejos. Algunos algoritmos generales basados en esta tendencia serían el seguimiento basado en kernel (seguimiento mean-shift), o el seguimiento de contorno (algoritmo de condensación). Actualmente este enfoque se lleva a cabo mediante el aprendizaje profundo.

Algunos ejemplos de trabajos basados en VTO se centran en modelar la apariencia del objetivo sin tener en cuenta el fondo [4], mientras que otros emplean un modelo de descomposición visual [5].

FIGURA 1.3-1: Ejemplo de seguimiento de personas sin considerar el fondo en [4].

1.3.2. Enfoque de seguimiento basado en LiDAR

A pesar de las ventajas de la alta resolución de las cámaras, a menudo los entornos presentan variaciones en la iluminación y puntos de vista que provocan una pérdida de precisión. Una alternativa al VTO que evita estos problemas, es el enfoque de seguimiento basado en la tecnología LiDAR, que consiste en un sistema láser de medida de distancias que permite realizar un barrido 3D del entorno que lo rodea. Hasta hace poco este tipo de tecnología no estaba al alcance de todos debido a su alto coste, pero actualmente se posiciona como uno de los elementos clave en los vehículos autónomos.

Algunos trabajos han seguido la línea de emplear modelos 3D ya adquiridos para realizar el seguimiento de los objetos [6]. Aunque también existen soluciones que no se basan en modelos, sino que detectan y rastrean los objetos dinámicos con el LiDAR y los segmentan mediante un enfoque bayesiano [7].

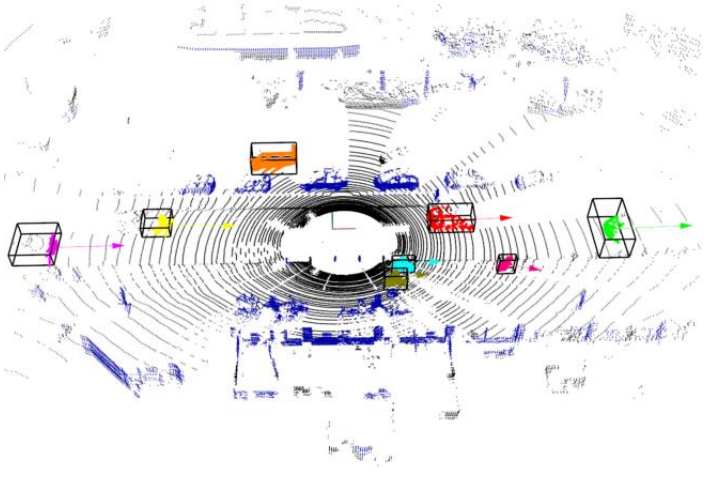

FIGURA 1.3-2: Seguimiento de objetos descrito en [7].

Una alternativa a estos estudios se basa en modelar los objetos como una nube de puntos en lugar de como una forma geométrica, empleando el algoritmo ICP (*Iterative Closest Points*) [8]. Sin embargo, este método requiere de una buena alineación inicial de puntos para evitar degradarse.

Para detectar y seguir objetos en movimiento en casos más complejos, otros autores proponen un sistema de seguimiento de rejilla de ocupación basado en partículas [9].

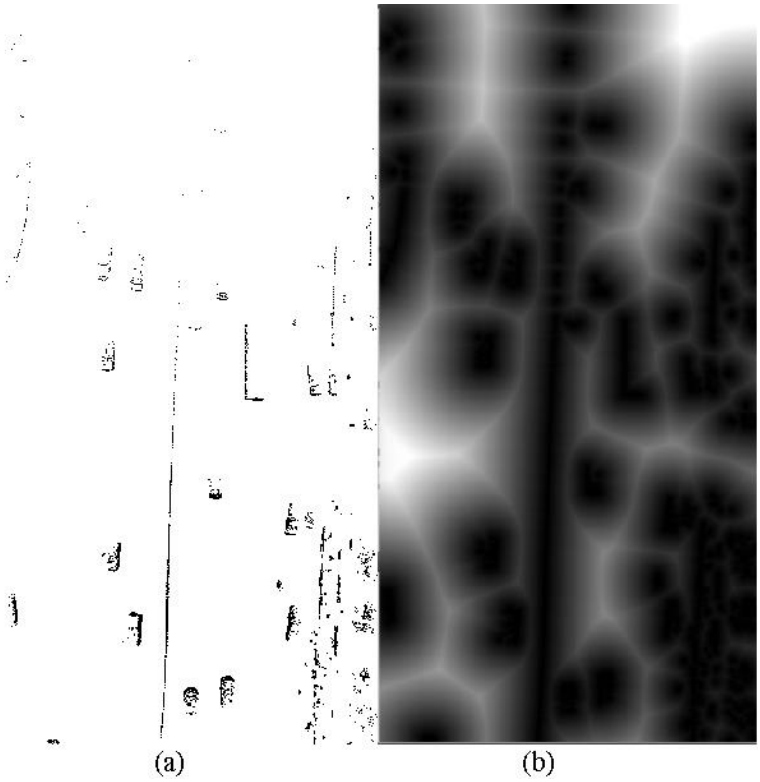

FIGURA 1.3-3: Detección de objetos basados en una rejilla en [9].

1.3.3. Enfoque de seguimiento basado en LiDAR y cámaras

Aunque en comparación con la cámara, el LiDAR sea robusto frente a la iluminación y presente un campo de visión horizontal más grande, tiene una baja resolución que puede resultar un inconveniente en la clasificación de objetos.

Con el objetivo de obtener la información más precisa de los sensores, puede interpretarse un tercer enfoque de seguimiento basado en un sistema híbrido que utiliza tanto LiDAR como cámaras, aprovechando las ventajas de ambas tecnologías. Un ejemplo de esta fusión de datos es el trabajo presentado en [10], que emplea los puntos obtenidos por el láser para asignar a cada píxel de la imagen una posición 3D.

Otros estudios utilizan un algoritmo de detección, seguimiento y fusión de obstáculos para reconstruir el entorno del vehículo empleando un UKF (*Unscented Kalman Filter*) [11]. Este filtro permite gestionar un número variable de observaciones, con lo que se consigue abordar correctamente el desafío combinado de seguimiento y fusión.

Siguiendo otra línea de estudio, en [12] se propone un algoritmo de seguimiento de objetos 3D utilizando un LiDAR, una cámara RGB y sensores de navegación inercial, INS (GPS/IMU) para obtener la trayectoria del objeto, la estimación de su velocidad actual y la predicción de posición en el sistema de coordenadas global.

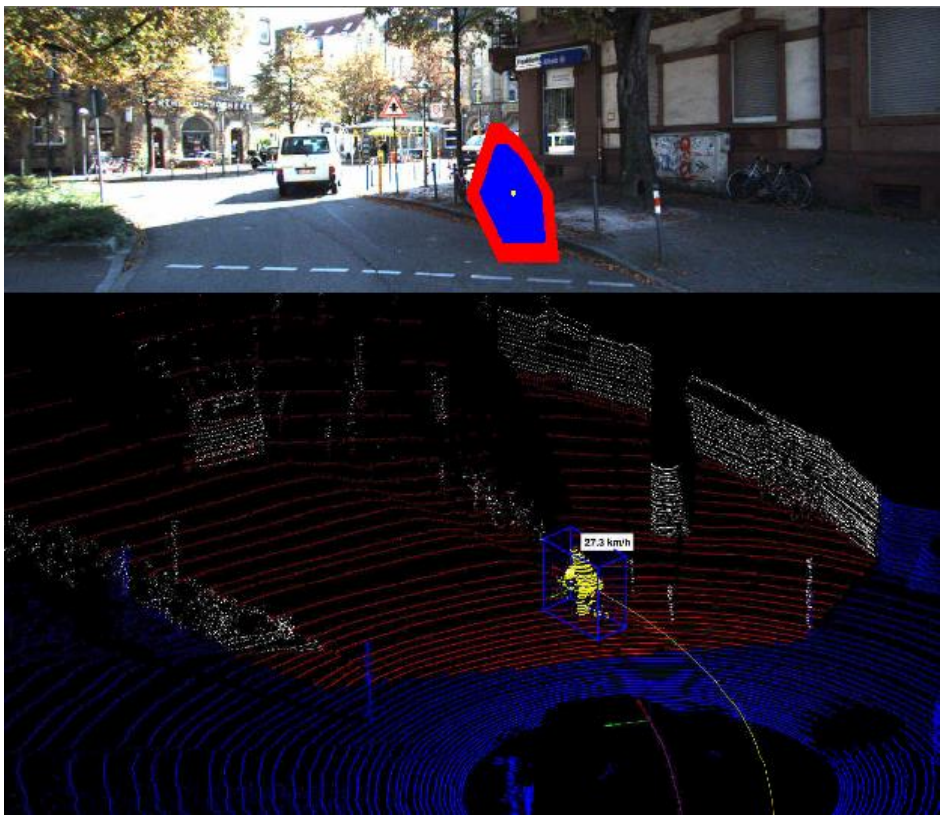

FIGURA 1.3-4: Método de seguimiento propuesto en [12].

1.4. Objetivos del trabajo

El objetivo principal del presente trabajo es comprender e implementar los algoritmos de precisión-tracking para diseñar un sistema de seguimiento de la posición y velocidad de los objetos de una escena urbana en tiempo real.

Se utilizará la plataforma ROS, y se hará uso de sus herramientas, como rviz, para la visualización 3D. Para simular el entorno de trabajo se empleará la plataforma de simulación VREP, donde se integrarán los sensores del vehículo (LiDAR 3D y cámara estéreo) y los objetos de la escena urbana: principalmente coches y personas (que serán los objetos donde se centre el seguimiento), así como otros de decoración, como la carretera, árboles o señales, para recrear la escena de manera fiel.

Para realizar el seguimiento, previamente será necesario efectuar la detección de los objetos de interés. Para ello se empleará la información procedente de los sensores obtenida como una fusión entre los puntos reflejados en el láser y el color obtenido de la cámara empleando el paquete *but_velodyne* para su calibración conjunta.

Esta información será convertida a nubes de puntos gracias a la librería PCL. Las escenas suelen contar con un gran número de puntos que representan cada uno de los elementos del entorno, sin embargo, cuanto mayor es el número de puntos, mayor es el esfuerzo de cómputo, por lo que será necesario filtrar el conjunto total para eliminar las zonas comunes que no son de interés para la detección, como la carretera.

Es siguiente objetivo es la segmentación 3D de la nube de puntos filtrada para poder clasificarla en los clústeres que forman los objetos, para lo cual se empleará la forma y el color (distintivo de cada objeto). Los clústeres se extraerán basándose en su distancia euclídea empleando la estructura KD-Tree.

Una vez que los objetos son detectados se aplicará el filtro de Kalman para poder realizar la asociación de datos y poder introducir clústeres que pertenecen al mismo objeto en la implementación del algoritmo de precisión-tracking para su seguimiento.

El objetivo final será realizar una comparativa de los resultados obtenidos en el seguimiento de la velocidad ante diferentes escenas con el filtro de Kalman y la técnica de precision-tracking.

Resumiendo, los principales objetivos del trabajo son:

- Creación de entornos en VREP y fusión multisensorial.
- Estudio e implementación de algoritmos para el filtrado y la segmentación 3D de la escena.
- Estudio e implementación del filtro de Kalman para el seguimiento de objetos y la asociación de datos.
- Estudio e implementación de los algoritmos de precisión-tracking.
- Pruebas en diferentes tipos de escenas para la evaluación de los resultados de seguimiento.

1.5. Estructura de la memoria

En este apartado se realizará un breve resumen de cómo se encuentra organizada la memoria explicando a grandes rasgos lo que el lector encontrará en cada capítulo:

- **Capítulo 1. Introducción**. Este es el capítulo en el que nos encontramos, donde se presenta una breve presentación acerca del desarrollo de los vehículos autónomos y el contexto en que se sitúa el presente trabajo. Además, se expone el estado del arte y los objetivos que se van a tomar como meta para su realización.

- **Capítulo 2**. **Herramientas**. En este capítulo se aborda una descripción detallada sobre los elementos de hardware (LiDAR 3D VLP-16, cámara estéreo Bumblebee® XB3, GPS Hiper Pro Topcon) y software (ROS, VREP, PCL) que son necesarios para el desarrollo del trabajo.

- **Capítulo 3**. **Arquitectura del Sistema**. En este capítulo comienza el desarrollo del trabajo. Se explicará de forma general cada una de las etapas necesarias para la detección y seguimiento de objetos en las que se ahondará en capítulos posteriores. Además, se detalla la implementación llevaba a cabo para realizar la fusión sensorial.

- **Capítulo 4. Detección y Seguimiento de Objetos**. Este es el capítulo principal del proyecto, donde se desarrolla en detalle cada una de las etapas necesarias para realizar la segmentación de objetos y su posterior seguimiento empleando el algoritmo de precision-tracking.

- **Capítulo 5. Sistema real.** En este capítulo se indican los cambios en la arquitectura descrita en simulación, para poder implementar un caso real.

- **Capítulo 6. Evaluación de Resultados.** En este capítulo se evaluará el rendimiento del sistema de seguimiento mediante precisión-tracking comparándolo con el seguimiento mediante filtro de Kalman a partir de diferentes métricas en diferentes escenas de objetos.

- **Capítulo 7. Conclusiones y Trabajos Futuros**. Como su propio nombre indica se mostrarán las conclusiones extraídas de la realización del trabajo, y se propondrán posibles trabajos atendiendo a las posibilidades de los elementos utilizados.

- **Planos.** En este capítulo se muestra parte del código desarrollado para la implementación del trabajo.

- **Pliego de Condiciones**. Este capítulo se exponen las herramientas necesarias para el desarrollo del trabajo y sus características principales.

- **Presupuesto.** En este capítulo se estima el presupuesto necesario para la realización del trabajo.

- **Capítulo 10. Manual de Usuario**. En este capítulo se explicará en detalle el proceso de instalación y de ejecución para poder implementar de manera externa el sistema descrito.

- **Bibliografía**. El último capítulo recoge cada una de las referencias a otros trabajos, libros o manuales que han sido consultados.

Capítulo II

HERRAMIENTAS

En este apartado se realizará una breve introducción y resumen de las principales funcionalidades de los elementos software y hardware que han sido empleados para el desarrollo del presente trabajo.

2.1. ROS

ROS, acrónimo de Robot Operating System, es una plataforma flexible de desarrollo de software en código abierto para robótica que tiene como objetivo simplificar el comportamiento robótico y con ello conseguir un software de propósito general [13].

FIGURA 2.1-1: Logotipo de ROS

Proporciona los servicios de un sistema operativo como la abstracción de hardware, el control de dispositivos de bajo nivel, el intercambio de mensajes entre procesos o la administración de paquetes. A pesar de no ser una plataforma en tiempo real, es posible la integración de código para este fin.

Una de sus grandes fortalezas es que cualquier usuario puede iniciar su propio repositorio de código ROS en sus servidores, sobre el que mantiene la propiedad, sin permisos, con la posibilidad de hacer público su trabajo y poder obtener reconocimiento por sus logros y realimentarse con las mejoras propuestas por la comunidad.

ROS se construyó desde la necesidad de un entorno de colaboración no limitado por parte de la comunidad investigadora en robótica, de modo que expertos en diferentes ámbitos de la robótica pudieran compartir sus trabajos y colaborar. La idea nace en la Universidad de Stanford a mediados de los 2000 con la creación de STAIR (Stanford AI Robot) y el programa PR (Personal Robots), pero no es hasta 2007, cuando Willow Garage (importante incubadora de empresas y laboratorio de investigación robótica) llevó a cabo su implementación.

Podemos enumerar algunas de las principales ventajas de ROS:

- Diseño distribuido y modular: puede emplearse desde proyectos sencillos a complejos, escogiendo qué módulos utilizar y cuáles implementar. Además, su naturaleza distribuida hace que la comunidad agregue un gran valor añadido con sus aportaciones de paquetes, permitiendo la integración con controladores hardware, herramientas de desarrollo o bibliotecas externas.

- Comunidad activa y colaborativa: la comunidad de ROS aúna un gran número de usuarios de todo el mundo, procedentes principalmente del sector de la investigación, con un aumento creciente en robótica industrial y de servicios, permitiendo contactar y colaborar con los principales desarrolladores de la plataforma. Además, es una comunidad activa que participa desde la elaboración de la documentación colaborativa hasta en el sitio web de ROS para preguntas y respuestas (*ROS Q&A*).

- Licencias permisivas: se encuentra autorizado bajo la licencia estándar BSD de tres cláusulas[1], que es una licencia abierta muy permisiva que permite su reutilización en productos comerciales y de código cerrado. Además, los paquetes de la comunidad incluyen otro tipo de licencias (Apache 2.0, GPL, MIT, entre otras) para que el usuario pueda identificar si un paquete puede satisfacer sus necesidades.

2.1.1. Conceptos Básicos

A nivel de sistema de archivos, podemos encontrarnos en ROS principalmente con dos conceptos:

- Paquetes: es la unidad más pequeña de compilación y la principal para la organización de software en ROS. Puede contener procesos de ejecución (nodos), bibliotecas, conjuntos de datos o archivos de configuración, entre otros. Permiten que el software pueda reutilizarse de manera sencilla porque contienen las funcionalidades necesarias sin llegar a ser muy pesadas.

- Metapaquetes: son paquetes especializados que sirven para referenciar de forma flexible a un conjunto de paquetes relacionados agrupados. No instalan archivos y no contienen códigos ni elementos que suelen encontrarse en un paquete.

[1] Para más información acceder a: https://opensource.org/licenses/BSD-3-Clause

El nivel de gráfico de computación es una red peer-to-peer (P2P) de los procesos de ROS. Sus conceptos básicos, que se implementan en el repositorio *ros_comm*, son los siguientes:

- Nodos: procesos individualizados que realizan cálculos reduciendo la complejidad del código. Todos los nodos en ejecución tienen un nombre identificativo único en el sistema y un tipo que indica el paquete del que procede. Un nodo ROS se escribe utilizando una biblioteca cliente ROS, como *roscpp* (para código en C++) o *rospy* (para código en Python). Puede monitorizarse y obtenerse información de los nodos en ejecución con la herramienta *rosnode*.

- Topics: canales de comunicación empleados por los nodos para intercambiar mensajes siguiendo una estructura de publicación/subscripción. Un nodo envía un mensaje publicándolo en un topic, o recibe su información subscribiéndose a él. Los subscriptores y publicadores se encuentran desacoplados, por lo que varios nodos pueden publicar y subscribirse al mismo topic de forma simultánea, aunque cada topic solamente admite datos de un único tipo. La herramienta *rostopic* permite inspeccionarlos.

- Mensajes: los nodos se comunican entre sí mediante la publicación de mensajes a través de topics. Un mensaje es una estructura de datos simple, que comprende distintos tipos de datos: desde estándar (entero, punto flotante, booleano, etc.), hasta estructuras y vectores anidados (como en el lenguaje C). Los nodos también pueden intercambiar un mensaje de solicitud y respuesta como parte de una llamada de servicio ROS.

- Servicios: a pesar de la flexibilidad de la comunicación entre topics, no es adecuada para las interacciones de solicitud/respuesta que suelen requerirse en un sistema distribuido para determinados eventos. Estas interacciones en ROS se realizan mediante servicios. Un nodo proveedor ofrece un servicio y un nodo cliente llama al servicio enviando el mensaje de solicitud y esperando la respuesta. La herramienta que permite gestionar los servicios es *rosservice*.

- Nodo maestro: coordina y sincroniza todo el sistema de nodos, topics y servicios. Su objetivo es permitir la comunicación entre nodos de igual a igual (P2P). También proporciona el servidor de parámetros, que es un diccionario compartido y multivariable que utilizan los nodos para almacenar y recuperar parámetros en tiempo de ejecución. ROS necesita un nodo maestro en ejecución continuamente, para ello, debe teclearse en un terminal: *roscore*.

- Bags: formato de archivo para guardar y reproducir datos de mensajes de ROS. Es un mecanismo importante para almacenar datos que pueden ser difíciles de recopilar pero que son necesarios para desarrollar y probar algoritmos. La herramienta que permite gestionar los bags es *rosbag*.

2.1.2. Infraestructura de comunicaciones

ROS es una plataforma distribuida de procesos (*nodos*) que posibilita que los ejecutables se diseñen individualmente y se acoplen de manera flexible en tiempo de ejecución. Estos procesos pueden agruparse en paquetes o pilas (*stacks)* que pueden compartirse y distribuirse fácilmente.

En el nivel más bajo, ROS ofrece una interfaz de intercambio de mensajes que proporciona comunicación entre procesos y se conoce como "middleware", que permite el intercambio anónimo de mensajes a través de un mecanismo de publicación/subscripción. Como el sistema es asíncrono, los datos pueden guardarse y reproducirse sin modificar el código, pero como en muchas ocasiones se requiere una comunicación síncrona, ROS proporciona esta capacidad mediante los *servicios*.

El nodo maestro almacena la información de registro de topics y servicios. Los nodos al comunicarse con el maestro pueden recibir información sobre otros nodos registrados y realizar conexiones entre ellos. El Maestro también realizará devoluciones de llamada a estos nodos cuando cambie la información de registro, permitiendo crear conexiones dinámicas según se ejecutan nuevos nodos.

Los nodos se conectan entre sí directamente; el maestro solo proporciona información de búsqueda, como un servidor DNS. Los nodos que se suscriben a un topic solicitarán conexiones de los nodos que publican ese topic, y establecerán esa conexión mediante un protocolo de conexión acordado. El protocolo más común utilizado en un ROS se llama TCPROS, que usa conectores estándar TCP/IP.

Esta arquitectura permite la operación desacoplada, donde los nombres son el medio principal por el cual se pueden construir sistemas más grandes y más complejos. Cada biblioteca cliente de ROS admite la reasignación de nombres en línea de comandos, por lo que un programa compilado puede reconfigurarse en tiempo de ejecución para operar en una topología diferente. En la siguiente imagen podemos ver un esquema de la arquitectura descrita:

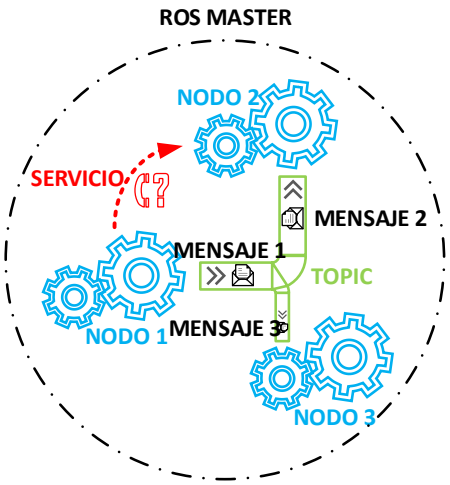

FIGURA 2.1-2: Esquema representativo de la arquitectura de ROS

2.1.3. Herramientas

Una de las principales características de ROS es su gran conjunto de herramientas de desarrollo, que permiten entre otras cosas la depuración y visualización del estado del sistema. Estas herramientas hacen uso de utilidades gráficas y comandos que simplifican el desarrollo. Entre las principales que emplearemos en el presente trabajo, destacan:

- Rviz: proporciona una visualización 3D de varios tipos de datos de sensores y de robots descritos mediante URDF. Permite visualizar varios tipos de mensajes comunes de ROS, como escaneos láser, nubes de puntos 3D e imágenes de cámara. También emplea información de la biblioteca tf para mostrar los datos del sensor en un marco de coordenadas común, junto con el modelo del robot. Es útil para ver problemas de desalineaciones entre sensores o imprecisiones en el modelo del robot.

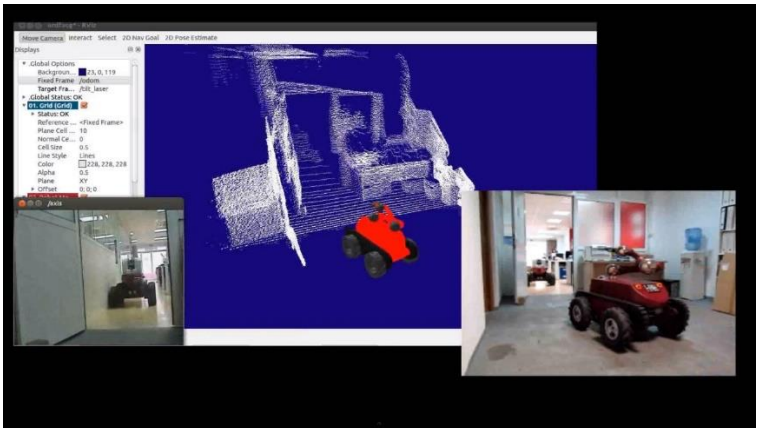

FIGURA 2.1-3: Ejemplo de interfaz RVIZ.

- Rqt: es un marco de referencia basado en Qt que permite crear interfaces personalizadas a través de la extensa biblioteca de pluggins que incorpora. Además, puede crear y añadir nuevos componentes mediante complementos propios. Algunos de sus pluggins más importantes son:

  - Rqt_graph: inspecciona y visualiza el estado del sistema en vivo, mostrando los nodos y sus conexiones.
  - Rqt_plot: supervisa cualquier variable representable con respecto al tiempo.
  - Rqt_topic: supervisa cualquier topic del sistema.
  - Rqt_publisher: permite publicar mensajes en los topics.
  - Rqt_bag: permite crear y acceder gráficamente a los bags. Puede registrar sus datos, reproducir sus topics y visualizar su contenido.

2.1.4. Otras características

- Características específicas: ROS proporciona también bibliotecas y herramientas comunes para poner en marcha robots de manera rápida. Cuenta con un conjunto de formatos de mensajes estándar de uso común en robótica, desde conceptos geométricos (posición, transformaciones), como mensajes de sensores (cámaras, IMU, láser) y datos de navegación (odometría, mapas). Además, proporciona herramientas para describir y modelar robots empleando el formato URDF (*Unified Robot Description Format*) mediante un documento XML que describe sus propiedades físicas.

- Sistemas operativos: ROS se ejecuta en plataformas basadas en Unix, principalmente en sistemas Ubuntu y Mac OS X, aunque la comunidad ROS ha estado contribuyendo con soporte para otras plataformas. Este es uno de los motivos por los que para el desarrollo del presente trabajo se empleará el sistema operativo Ubuntu, y más en concreto su versión 14.04.

- Distribuciones: son colecciones de paquetes con diferentes versiones consistentes que pueden instalarse. Similar a las distribuciones de Linux, suministran una base de código estable que facilita instalar un conjunto de software. Se organizan en distribuciones anuales, teniendo un soporte de 2 años las liberadas en años impares, mientras que las de años pares (denominadas LTS) un soporte de 5 años. De este modo cada versión LTS de ROS está ligada a una versión LTS de Ubuntu. En el presente trabajo se ha optado por emplear la distribución Indigo.

- Independencia de lenguaje: Permite implementarse con cualquier lenguaje de programación moderno como Python, C++ y Lisp. Además, dispone de bibliotecas experimentales en Java y Lua. El desarrollo de código para los nodos en este trabajo se ha realizado mediante C++.

- Integración con librerías externas: ROS proporciona integración entre diferentes librerías de código abierto, junto con un sistema de envío de mensajes para el fácil intercambio entre diferentes fuentes de datos. Entre las principales librerías con las trabajaremos en el presente trabajo destacan:

  - OpenCV: es la principal biblioteca de visión por computadora utilizada mundialmente. Proporciona algoritmos y utilidades comunes para la calibración de cámaras, el procesamiento de imágenes, la segmentación y el seguimiento.

  - PCL: es una biblioteca de percepción que se centra en el manejo y procesamiento de datos 3D e imágenes de profundidad. Proporciona algoritmos de filtrado y detección de características entre otros como veremos en más detalle en un apartado posterior.

2.2. VREP

Es un marco de simulación versátil y escalable con entorno de desarrollo integrado, que acelera la simulación distribuyendo la carga de la CPU sobre varios núcleos o varias máquinas. Ofrece diferentes técnicas de programación, permite incorporar controladores y funcionalidades en modelos de simulación, facilita la tarea de programadores y reduce la complejidad de implementación para los usuarios. [14].

FIGURA 2.2-1: Logotipo de VREP versión PRO EDU.

Se basa en una arquitectura de control distribuido, donde cada objeto/modelo puede controlarse individualmente a través diferentes técnicas de programación, lo que favorece su versatilidad para aplicaciones robóticas.

Puede emplearse para diversos fines, como la simulación de sistemas de automatización industrial, el monitoreo remoto, el desarrollo de algoritmos o como herramienta educativa para la robótica. Se puede utilizar como una aplicación independiente o integrarse fácilmente en una aplicación principal gracias a su API.

2.2.1. Arquitectura

Es un simulador personalizable gracias a una interfaz de programación de aplicaciones (API) elaborada. Soporta seis enfoques diferentes de programación mutuamente compatibles (pueden usarse al mismo tiempo, o conjuntamente):

- Scripts embedidos: la estructura de simulación principal es un script simple en lenguaje de programación Lua que, a partir de una escena de simulación, maneja la funcionalidad general, llamando a los scripts secundarios, que se adjuntan a objetos específicos en la escena y manejan una parte particular de la simulación.

- Extensiones (*add-ons*): son también compatibles a través de scripts Lua. Se pueden usar como funciones independientes, iniciarse automáticamente y ejecutarse en segundo plano, o llamarse como funciones. No son específicas de una simulación o modelo, sino que ofrecen una funcionalidad más genérica, vinculada al simulador.

- Complementos (*plugings*): se utilizan para la personalización del simulador, empleándose junto con scripts embedidos. Pueden extender la funcionalidad de un modelo u objeto de simulación, proporcionar una funcionalidad especial con una capacidad de cálculo rápida (los scripts suelen ser más lentos que los lenguajes compilados) o proveer una interfaz de comunicación.

- APIs de clientes remotos: permite la interacción externa con V-REP a través de una comunicación mediante socket. Lo componen servicios de servidor API remotos y clientes API remotos. El lado del cliente puede incorporarse en cualquier hardware y permite la función remota de llamadas, así como la transmisión rápida de datos. El modo de operación posibilita la llamada de funciones como bloqueo (esperará hasta que el servidor responda), o sin bloqueo (leerá los comandos transmitidos desde un búfer, o iniciará/detendrá un servicio de transmisión por el servidor).

- Nodos ROS: se implementan con un complemento que permite a ROS llamar a los comandos V-REP a través de servicios ROS o transmitir datos a través de los publicadores/suscriptores de ROS, que pueden habilitarse con una llamada de servicio o directamente desde V-REP a través de un comando de script embedido.

- Nodos BlueZero: permite que una aplicación externa se conecte a V-REP a través de BlueZero (plataforma similar a ROS enfocada en la comunicación y transporte de mensajes, que proporciona herramientas para la conexión de software en múltiples hilos, varios procesos o distintas máquinas).

La siguiente figura ilustra las diversas posibilidades de personalización en V-REP y sus alrededores:

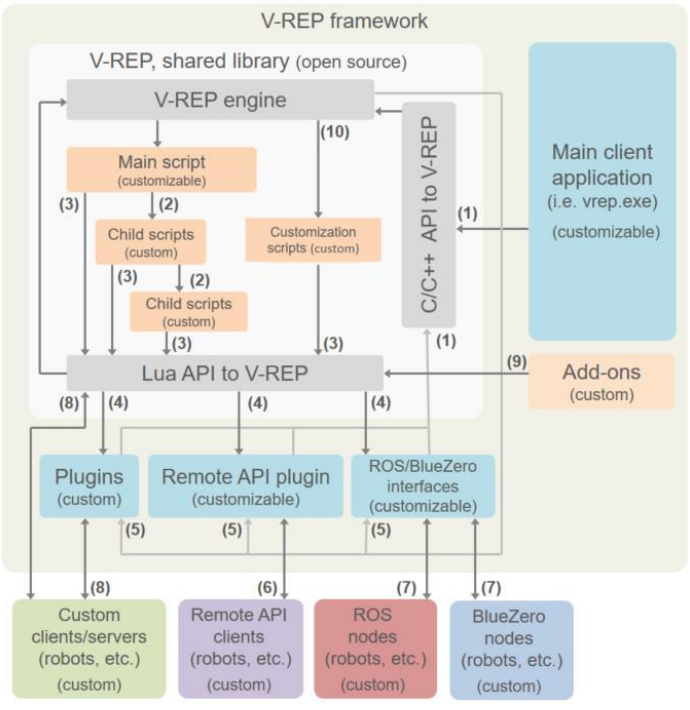

FIGURA 2.2-2: Framework de programación V-REP.

V-REP cuenta con varios tipos de interfaces como las APIs regular, remota y auxiliar y las interfaces de ROS y BlueZero. A la API regular sólo se accede desde el simulador, mientras que, al resto, se puede acceder desde cualquier aplicación externa o hardware.

La API auxiliar no es una interfaz como tal, sino una colección de funciones auxiliares que pueden integrarse y funcionar por sí mismas, mientras que la API remota permite controlar una simulación (o el simulador) desde una aplicación externa o un hardware remoto mediante la comunicación por socket de forma síncrona o asíncrona (estructura cliente-servidor).

La siguiente figura ilustra una descripción general de las diversas interfaces:

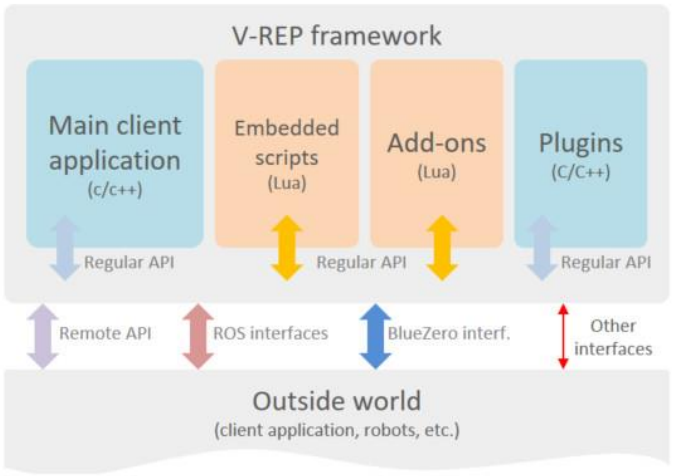

FIGURA 2.2-3: Interfaces de V-REP.

2.2.2. Funcionalidades

V-REP está diseñado como una arquitectura versátil con diversas funcionalidades relativamente independientes que pueden habilitarse de manera selectiva o cooperar conjuntamente. La funcionalidad de V-REP se relaciona con los objetos de escena específicos, o con los módulos de cálculo específicos que se describen a continuación.

2.2.2.1. Objetos de escena

Una escena de simulación o modelo contiene varios objetos agrupados en una jerarquía en forma de árbol. Algunos de los objetos con los que podemos encontrarnos son:

- Articulaciones: son elementos que unen dos o más objetos de una escena con al menos un grado de libertad (prismáticos, de revolución, tipo tornillo o esféricos) y que pueden funcionar en varios modos (fuerza/par o cinemática inversa, entre otros).

- Formas: son mallas triangulares, utilizadas para simulación y visualización de cuerpos rígidos. Son determinantes para realizar los cálculos.

- Sensores de proximidad: calculan la distancia mínima exacta entre su punto de detección y cualquier entidad detectable que interfiere con su volumen de detección, dando como resultado una operación continua y una simulación realista. Puede modelarse casi cualquier tipo de sensor (ultrasonidos, infrarrojo o láser, entre otros).

- Sensores de visión: son objetos visibles que renderizan los objetos de su campo de visión y activan la detección en función de umbrales especificados. Puede accederse al contenido de una imagen directamente a través de una API, permitiendo extraer información de una escena de simulación. Utilizan la aceleración de hardware para la adquisición de imágenes en bruto (OpenGL), lo que hace que funcionen más lento.

- Sensores de fuerza: representan enlaces rígidos entre formas, que pueden registrar fuerzas y pares aplicados. Su rigidez es condicional, por lo que pueden romperse al sobrepasar un umbral. Sólo funciona durante simulación si se habilita dinámicamente.

- Gráficos: se emplean para registrar, visualizar o exportar datos de una simulación. Los datos se registran como una secuencia que puede visualizarse en función del tiempo, o combinarse entre sí para mostrar gráficos X/Y o curvas 3D.

- Cámaras: permiten la visualización de escenas cuando están asociadas a una ventana gráfica. Son objetos visibles, por lo que se puede ver a través de ellos y mostrar una vista de lo que están observando.

- Luces: son objetos que te permiten iluminar una escena y que influyen directamente en las cámaras o los sensores de visión. Hay tres tipos diferentes de luz: omnidireccional, direccional o proyectores.

- Dummy: es el objeto más simple disponible. Es un punto de orientación que puede verse como un marco de referencia, y que son útiles al emplearse junto a otros objetos o módulos.

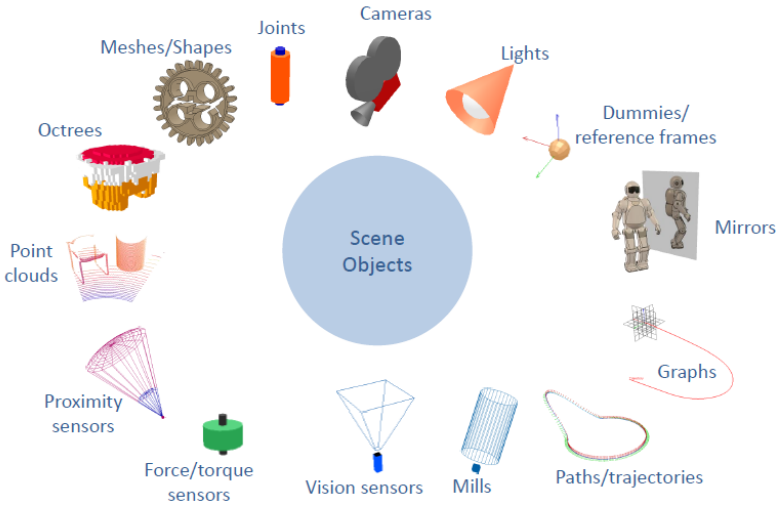

FIGURA 2.2-4: Ejemplos de objetos disponibles en VREP.

2.2.2.2. Módulos de cálculo

V-REP ofrece varios módulos de cálculo que pueden operar directamente en uno o varios objetos de escena. A continuación, se muestran los principales:

- Módulo cinemático: permite cálculos cinemáticos (directos/inversos) para cualquier tipo de mecanismo. También está disponible para aplicaciones externas a través de las funciones cinemáticas externas.

- Módulo dinámico: permite manejar el cálculo e interacción de la dinámica de un cuerpo rígido a través de cuatro motores de física: la biblioteca de física Bullet, el motor de Open Dynamics, el motor Vortex Dynamics y el motor Newton Dynamics. Las simulaciones dinámicas suelen basarse en aproximaciones, por lo que es importante no confiar en un único motor de física para validar los resultados.

VREP combina la cinemática y la dinámica para obtener el mejor rendimiento en función del escenario de simulación. Los cálculos dinámicos suelen ser imprecisos y lentos, por lo que siempre que sea posible es mejor realizar cálculos cinemáticos.

- Módulo de detección de colisiones: permite una rápida verificación de interferencias entre formas. Es independiente del módulo dinámico, ya que sólo detecta la colisión sin dar una respuesta (función dinámica).

- Módulo de cálculo de distancia entre mallas: permite cálculos rápidos de distancia mínima entre formas. Usa las mismas estructuras de datos que el módulo de detección de colisión. Es independiente del módulo dinámico, ya que sólo mide la distancia sin dar una respuesta (función dinámica).

- Módulo de planificación de ruta/movimiento: maneja tareas de planificación de ruta holonómica y no holonómicas[2]. Considera tanto el estado inicial y el objetivo, como las entidades de colisión. Suele emplearse junto con la cinemática inversa para buscar la mejor ruta posible.

Los módulos de cálculo se implementan de forma general sobre escenas o modelos de simulación. El objetivo de integrarlos en lugar de depender de bibliotecas externas es similar al de tener scripts embedidos: la mayoría de las simulaciones o modelos no requieren herramientas específicas, pero si un conjunto de herramientas básicas, que al integrarse hace que los modelos se vuelvan portátiles, distribuyéndose entre diferentes máquinas o plataformas en un único archivo.

[2] En robótica, el término holonómico, simplificando la definición, se refiere a cuando un robot es capaz de moverse instantáneamente a una posición sin necesidad de realizar otros movimientos intermedios.

2.3. PCL

La librería Point Cloud (PCL) es un proyecto abierto a gran escala para imágenes 2D/3D y procesamiento de nubes de puntos [15].

FIGURA 2.3-1: Logotipo de PCL.

Una nube de puntos es una estructura de datos que representa un conjunto de puntos en varias dimensiones (coordenadas geométricas XYZ) de una superficie muestreada, y que puede incluir una cuarta dimensión si se dispone de color. Las nubes de puntos se pueden adquirir a partir de sensores de hardware como cámaras estéreo, escáneres 3D o cámaras de tiempo de vuelo, o pueden generarse a partir de un programa informático sintéticamente.

PCL es una biblioteca de C++ moderna modelada para el procesamiento en 3D de nubes de puntos. Se basa en la biblioteca *Eigen* para las operaciones matemáticas, y en *FLANN* (biblioteca rápida para vecinos cercanos) para la búsqueda de puntos vecinos; además emplea punteros compartidos *Boost*. Incorpora una multitud de algoritmos de procesamiento 3D (filtrado, obtención de características, reconstrucción de superficies o segmentación, entre otras), que integran todas sus funcionalidades de forma compacta.

Al igual que ROS, utiliza la licencia BSD de 3 cláusulas y es un software de código abierto, siendo gratis para uso comercial y de investigación. Además, es un software multiplataforma compatible con Linux, MacOS, Windows y Android / iOS.

2.3.1. Módulos

Para simplificar el desarrollo, PCL se divide en una serie de bibliotecas menores, que se pueden compilar por separado, permitiendo su distribución en plataformas con limitaciones computacionales. A continuación, se muestran los principales módulos que van a emplearse en este trabajo (en algunos de ellos se entrará en más detalle en el apartado de implementación):

- Filtros (pcl_filters): contiene mecanismos de eliminación de ruido y valores atípicos (outliers) para aplicaciones de filtrado de datos de nubes de puntos 3D.

- Kd-tree (pcl_kdtree): proporciona la estructura de datos kd-tree, que permite realizar búsquedas rápidas en los puntos vecinos más cercanos al punto de análisis. Kd-tree (árbol de dimensión k) es una estructura de datos de partición de espacio que almacena un conjunto de puntos k-dimensionales en una estructura de árbol para realizar búsquedas de rango y de vecinos cercanos.

- Segmentación (pcl_segmentation): contiene algoritmos para segmentar una nube de puntos en distintos clústeres (subconjuntos de puntos espacialmente aislados dentro de la nube de puntos). Se emplea para procesar una nube de puntos compuesta de regiones espacialmente aisladas, que se divide para procesarse independientemente.

- Muestreo (pcl_sample_consensus): contiene métodos de muestreo como RANSAC y modelos de formas, que pueden combinarse libremente para detectar modelos específicos y sus parámetros en nubes de puntos.

- Entrada/salida (pcl_io): contiene clases y funciones para leer y escribir archivos de datos de nubes de puntos (PCD), así como capturar nubes de puntos con diferentes dispositivos.

- Visualización (pcl_visualization): permite prototipar y visualizar los resultados de los algoritmos que operan en datos de nubes de puntos 3D. Similar a OpenCV para mostrar imágenes 2D y para dibujar formas 2D básicas en la pantalla, la biblioteca permite representar y establecer propiedades visuales, dibujar formas 3D y su visualización mediante histogramas 2D entre muchas otras cosas.

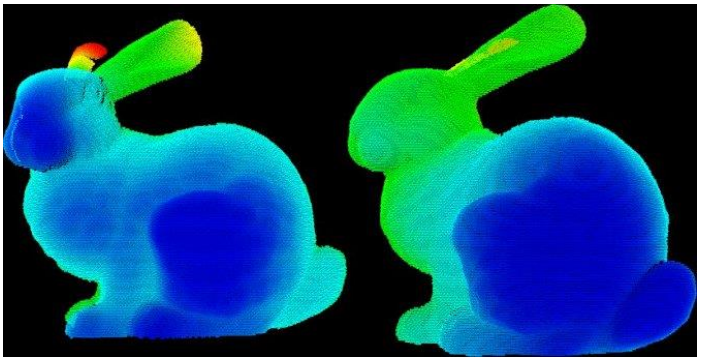

FIGURA 2.3-2: Ejemplo de visualización del módulo pcl_visualization

2.3.2. PCL-ROS

La filosofía de diseño de PCL se basa en que la mayoría de las aplicaciones que se ocupan del procesamiento de nubes de puntos se generen como un conjunto de bloques que se parametriza para lograr los resultados.

Basado en el diseño de otras bibliotecas de procesamiento 3D y de ROS, cada algoritmo de PCL está disponible como un bloque independiente que se puede conectar fácilmente con otros bloques de la misma manera que los nodos se conectan entre sí en ROS. Además, debido a que las nubes de puntos son grandes, para garantizar que no se copian en aplicaciones críticas se crean los nodelets, que son complementos cargables dinámicamente que se ven y funcionan como nodos ROS, pero en un solo proceso (como hilos simples o múltiples).

2.4. Sensores

Como el trabajo está orientado al ámbito de la navegación autónoma, es necesario repasar los principales sensores con los que el vehículo puede obtener la información de su entorno. Teniendo en cuenta que el número de sensores necesarios en un vehículo autónomo es superior a los que se van a presentar a continuación, nos centraremos en los tres sensores de mayor interés para obtener la información necesaria.

2.4.1. Sensor de distancia: LIDAR 3D

Los sensores de distancia activos son un elemento clave en la robótica móvil para la localización y modelado del entorno. Su principio de funcionamiento se basa en obtener la distancia a través de la velocidad de propagación de una onda emitida, y el tiempo que tarda desde que se emite hasta que se recibe (tiempo de vuelo):

$$\boxed{d = \frac{c_0 \cdot t}{2}}$$

Donde:

- c_0 es la velocidad de la luz para el caso del láser.
- t es el tiempo de vuelo.
- d es la distancia.

Los sensores láser tienen la ventaja de que obtienen mediciones de distancia muy precisas y más confiables que mediante otros sensores de distancia como los de ultrasonidos.

La tecnología LiDAR (acrónimo de *Light Detection and Ranging o Laser Imaging Detection and Ranging*) emplea un tipo de sensor que permite determinar la distancia desde un emisor hasta un objeto o superficie empleando un haz láser pulsado. Un diodo en su interior emite el rayo láser que se direcciona a través de una lente transmisora, golpea en el objetivo y parte de la luz se refleja en un fotodiodo tras pasar por una lente recpectora. Su sistema de funcionamiento es similar al de un radar, pero empleando láser en lugar de ondas de radio y utilizando más de un par emisor-receptor.

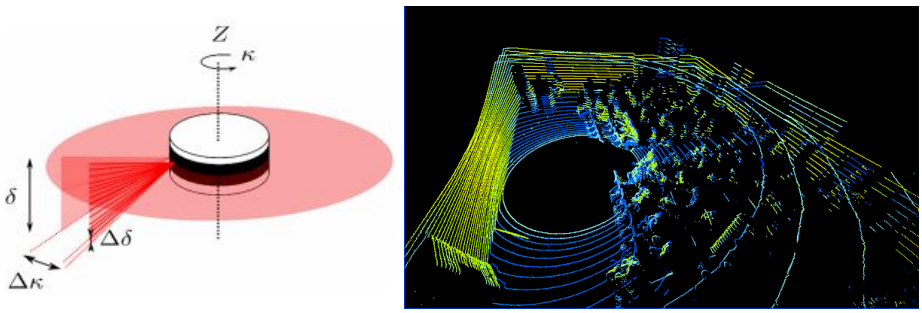

FIGURA 2.4-1: Esquema de funcionamiento de un LiDAR 3D.

El LiDAR 3D consiste en láseres rotativos apilados que obtienen la información del entorno desde diferentes ángulos, lo que permite obtener una nube de puntos. Cada capa de láseres es un canal que emite una señal que crea una línea de contorno, y que juntándose con las líneas del resto de canales genera una imagen tridimensional. Por tanto, cuanto mayor sea el número de canales, mayor resolución tendremos.

En el presente trabajo haremos uso del ***Velodyne LiDAR Puck* de 16 canales (VLP-16)** [16] que se caracteriza por ser el producto más pequeño y avanzado de la gama de LiDAR 3D de la empresa Velodyne, conservando las mediciones de reflectividad calibrada y en tiempo real a 360º.

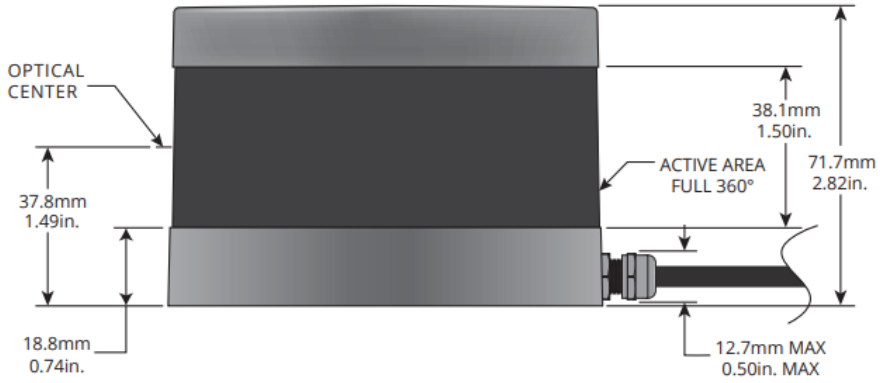

FIGURA 2.4-2: Esquema VLP-16.

En la siguiente tabla podemos ver algunas de sus características:

Características del sensor		**Características mecánicas/eléctricas**	
Canales	16	Consumo [W]	8
Rango máximo de medida [m]	100	Peso [g]	830
Precisión [cm]	3	Temperatura de operación [ºC]	(-10) a (+60)
Campo de visión vertical [º]	(+15) a (-15)	Grado de protección	IP67
Resolución angular vertical [º]	2	**Salida**	
Campo de visión horizontal [º]	360	Modo retorno simple (ptos/s)	300.000
Resolución angular horizontal [º]	0.1-0.4	Modo retorno dual (ptos/s)	600.000
Frecuencia de giro [Hz]	5-20	Conexión a internet [Mbps]	100

TABLA 2.4-1: Características técnicas VPL-16.

2.4.2. Sensor de visión: Cámara

La visión artificial o por computador se basa en capturar la información visual del entorno físico para extraer características relevantes visuales. El dispositivo esencial para obtener este tipo de información es la cámara, y uno de sus componentes principales es el sensor de visión, que forma parte del sistema de captación de la imagen, y se encarga de convertir las ondas de luz recibida en señales eléctricas, gracias a sus componentes fotosensibles. Estas señales son procesadas y convertidas en las imágenes que vemos.

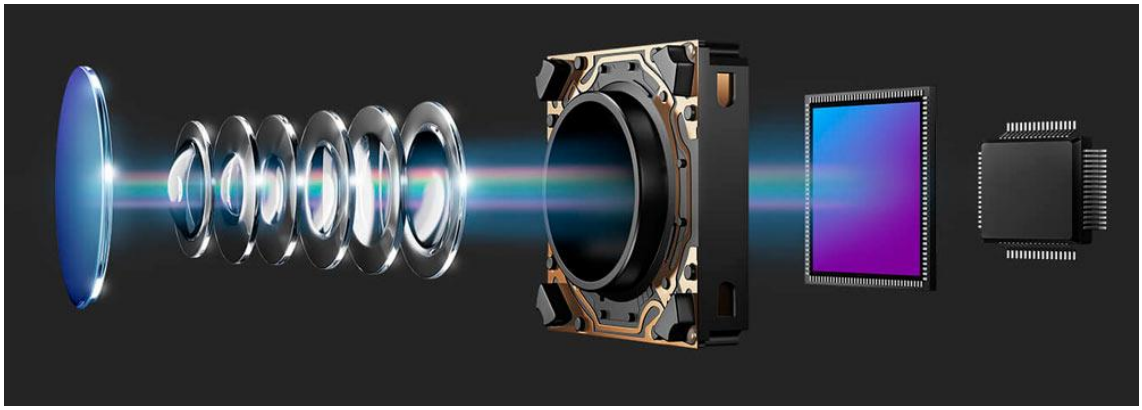

FIGURA 2.4-3: Imagen de una cámara y sus componentes.

Principalmente podemos encontrarnos con dos tipos de tecnologías de sensores: CCD (*Charge-Coupled Device*) que se basa en condensadores que acumulan la carga de los fotones incidentes y la transfieren a otros condensadores; y CMOS (*Complementary Metal-Oxid-Semiconductor*) que integra transistores MOSFET formando fotodiodos que obtienen un voltaje proporcional a los fotones incidentes. Las principales diferencias entre CCD y CMOS son que en el primer tipo existe una transferencia de carga eléctrica y que en el segundo los píxeles son tratados de manera individual.

Existen diferentes tipos de cámaras, pero en robótica es ampliamente utilizada la cámara estéreo debido a las ventajas que ofrece respecto a otros tipos de cámara como las de modelo *pinhole*. La cámara estéreo se basa en la visión binocular humana para capturar 2 imágenes que se procesan con el objetivo de obtener la distancia y profundidad de los elementos que constituyen la imagen, y así poder representar una imagen 3D. Uno de sus puntos críticos es el correcto alineamiento de los pixeles de la imagen de una cámara con la otra.

En el presente trabajo haremos uso de la **cámara estéreo Bumblebee® XB3** [17] que cuanta con 3 sensores CCD y dos líneas base (distancia entre cámaras) diseñados para dar flexibilidad y precisión. Cuenta con una línea base ampliada para obtener mejores resultados en largas distancias, mientras que la línea base pequeña mejora en el rango cercano.

FIGURA 2.4-4: Cámara estéreo Bumblebee® XB3.

En la siguiente tabla podemos ver algunas de sus características:

Especificaciones	
Sensor de imagen	3 sensores CCD Sony ICC445 1/3''
Línea base [cm]	12
	24
Distancia focal de lente [mm]	3.8 (66º HFOV)
	6 (43º HFOV)
Conversor A/D [bits]	12
Consumo (a 12V) [W]	4
Peso [g]	505
Tamaño de imagen [pixeles]	1280x960
Temperatura de operación [ºC]	(0) a (+45)
Interfaz	IEEE-1394b

TABLA 2.4-2: Características técnicas Bumblebee XB3.

2.4.3. Sensor de posicionamiento: GPS

Uno de los objetivos más importantes de un vehículo autónomo es mantener su posicionamiento y guiado en todo momento. Esta información en exteriores se obtiene gracias al GPS (*Global Positioning System*).

El GPS es un sistema que permite obtener la posición en cualquier punto de la Tierra con elevada precisión. Su funcionamiento se basa en la trilateración gracias a la red de 24 satélites que orbitan la Tierra a una altura de 20.180 km cubriendo toda su superficie. El receptor GPS recibe la señal procedente de los satélites indicando su posición y hora, y obtiene el tiempo que tarda en llegar la señal para calcular mediante trilateración la distancia de cada satélite al punto de medición.

Matemáticamente, con 4 satélites sería suficiente para determinar la posición exacta en la Tierra, ya que el lugar geométrico de los puntos del espacio que equidistan de cada satélite es una esfera, y, por tanto, es la intersección de 4 esferas la que permite obtener un punto. Sin embargo, en una aplicación de navegación autónoma el posicionamiento es crítico, y se necesitan precisiones elevadas que se consiguen con la detección de un mayor número de satélites.

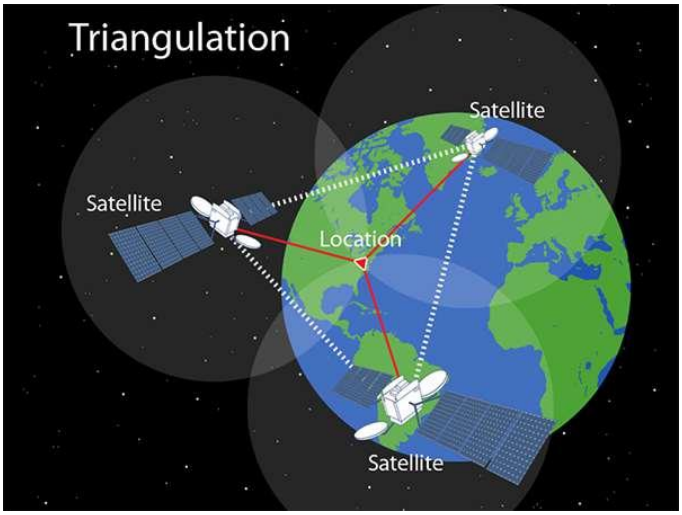

FIGURA 2.4-5: Imagen esquemática del proceso de trilateración de un GPS.

Para obtener el posicionamiento global se ha utilizado el **GPS HIPer Pro de Topcon**, [18] que se trata de un receptor GPS de doble frecuencia que proporcionaa un sistema cinemático en tiempo real (RTK) preciso. Incorpora tecnología de seguimiento por satélite GPS + GLONASS[3], lo que da una mayor cobertura satélite y rendimiento.

En la siguiente tabla podemos ver algunas de sus características:

Especificaciones	
Tipo de receptor	Euro-112T (HGGDT)
Canales estándar	20 (GPS, Diferencial y GLONASS)
Tiempo de operación [h]	+14
Consumo [W]	<4.2
Tipo de antena	UHF de montaje central
Comunicación inalámbrica	Bluetooth
Puertos de comunicación	x2 (RS2R2)
Frecuencia de salida [Hz]	20

TABLA 2.4-3: Características técnicas GPS HIPer Pro de Topcon.

[3] GLONASS es un Sistema Global de Navegación por Satélite administrado por Rusia y que es homólogo al sistema GPS de EEUU y al sistema Galileo de Europa.

Capítulo III

ARQUITECTURA GENERAL DEL SISTEMA

En este capítulo se mostrará una descripción general de la arquitectura que va a tener el sistema que se va a implementar, así como la información disponible procedente del entorno del vehículo y la fusión de los sensores para poder realizar la detección y seguimiento de los obstáculos.

3.1. Entorno de trabajo

Para poder comprender cada una de las etapas que intervienen en el sistema de seguimiento, resulta eficaz su representación visual mediante un esquema acompañado de una breve explicación:

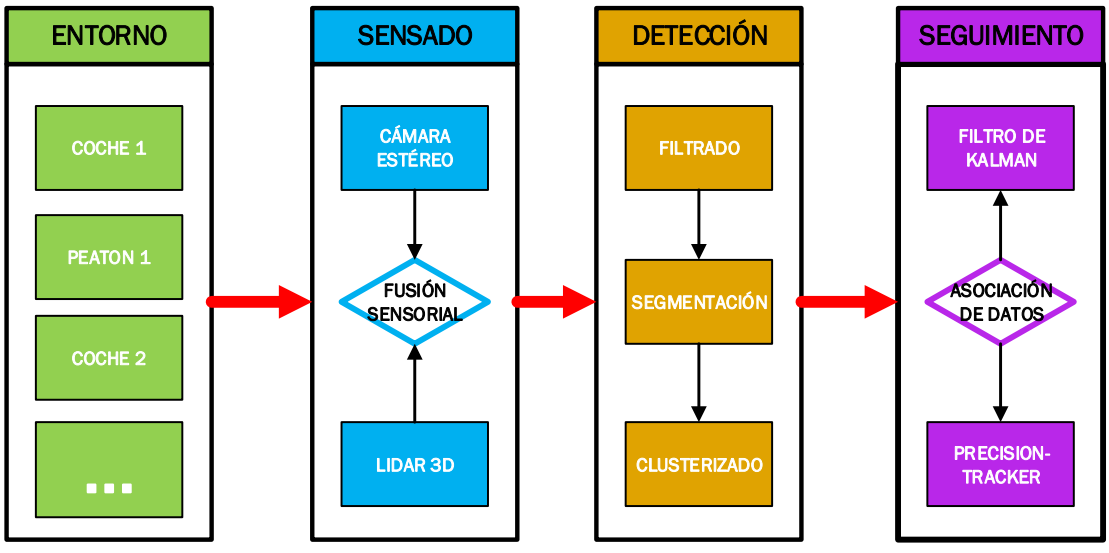

FIGURA 3.1-1: Arquitectura general del sistema**.**

El esquema que se muestra en la figura superior sigue el paradigma de seguimiento por detección (*tracking-by-detection*) que se utiliza comúnmente e implica que la detección y el seguimiento se realizan de manera sucesiva.

En la etapa de detección se toman los datos en bruto de la escena obtenidos por los sensores y se realiza una segmentación para dividir los objetos. Con esto se consigue centrar los esfuerzos computacionales en los objetos que se quieren seguir, eliminando la información que no resulta de interés. A continuación, se realiza una clasificación ajustando el objeto a un modelo establecido.

Una vez que los objetos son detectados e identificados, a cada objeto se le asigna un filtro de estimación de estado para poder predecir el estado siguiente del objeto en base a un modelo de movimiento. Con el objetivo de realizar una estimación eficaz de la evolución de los estados de cada objeto se empleará un filtro bayesiano como es el filtro de Kalman.

Como habitualmente en una escena de tráfico tenemos varios objetos a la vez que siguen diferentes trayectorias, el siguiente paso es determinar qué objeto pertenece a cada trayectoria en cada uno de los frames. La asociación de datos resulta un paso crítico para determinar que cada objeto se encuentre identificado de manera única. Para realizar esta etapa de nuevo haremos uso del filtro de Kalman.

Una vez realizada la asociación de datos entre cada uno de los frames, el último paso consiste en la gestión del seguimiento, para lo cual se implementará el algoritmo de precision-tracking.

La comunicación entre las diferentes etapas es fundamental en el sistema, ya que permite la transmisión de los flujos de datos. Esta comunicación se realiza mediante la arquitectura formada por nodos y topics de ROS. Haciendo uso de la herramienta rqt podemos ver un esquema de conexión de la arquitectura de ROS y poder relacionarlo con las etapas descritas anteriormente:

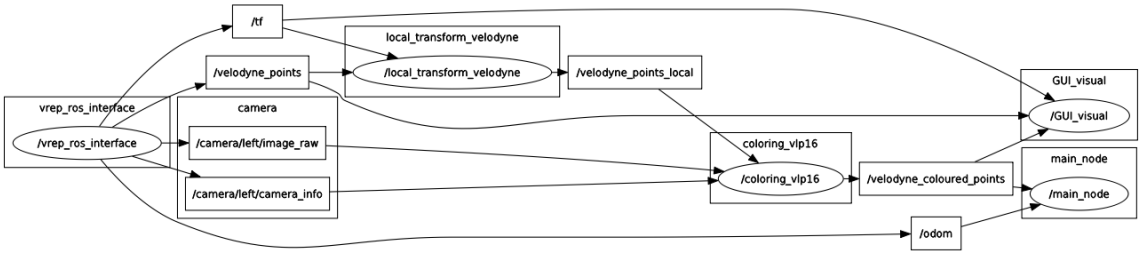

Figura 3.1-2 Arquitectura de conexión de topics y nodos.

En la figura anterior podemos ver que los nodos vienen representados por elipses, mientras que los topics por rectángulos rodeando el nombre correspondiente. Cuando de un nodo sale una flecha que termina en un topic indica que lo está publicando, mientras que si el topic va a parar a un nodo indica que se está suscribiendo a él.

El nodo origen es */vrep_ros_interface* que es generado por la plataforma VREP encargada de ejecutar la simulación. En este nodo se publicarán los topics del LiDAR (*/velodyne_points*) y de la cámara (*camera/left/image/raw* y */camera/left/camera_info*).

Las coordenadas globales del LiDAR se transformarán a las coordenadas locales de la cámara mediante el topic /*tf* y el nodo */local_transform_velodyne* para poder realizar la fusión sensorial en el nodo */coloring_vlp16* que publicará una nube de puntos en color en el topic */velodyne_coloured_points*, al cual se suscribirá el nodo principal de la aplicación (*/main_node*) donde se realizará la detección y el seguimiento. A estos mismos topics se suscribirá el nodo */GUI_visual* que permitirá la visualización de los datos en la plataforma rviz de ROS.

3.2. Entorno de adquisición de datos

Como ya se ha mencionado, VREP es la plataforma sobre la cual se encuentra construido el entorno donde van a ser realizadas las simulaciones. Para poder hacer una comparación visual con el entorno real, se ha implementado un mapa que representa el Campus Externo de la Universidad de Alcalá, donde se han añadido diferentes elementos que ofrece la plataforma para desarrollar un entorno más realista: como formas de los edificios, carreteras, señales de tráfico o los propios árboles.

En la siguiente figura podemos ver una comparativa entre el entorno desarrollado en VREP y el mapa real que representa, para comprobar las similitudes visuales entre ambos:

FIGURA 3.2-1: Entorno de simulación VREP (izquierda) y mapa real que representa (derecha).

Para poder realizar una simulación completa del entorno será necesario implementar los modelos tanto de vehículos y peatones que formarán parte de la escena real, como de los sensores que lleva abordo el vehículo. En la siguiente imagen pueden apreciarse varios modelos de objetos:

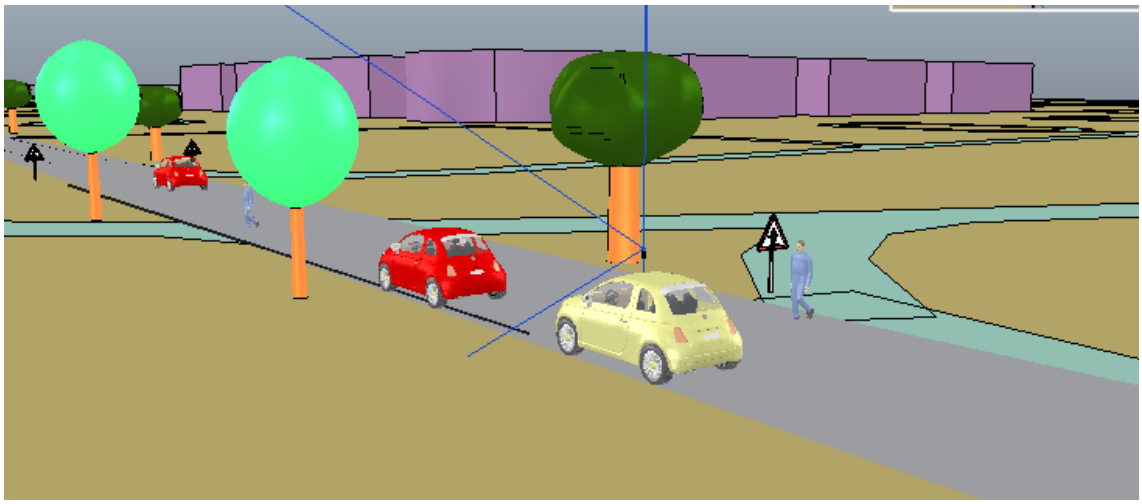

FIGURA 3.2-2: Ejemplo de escena en VREP con varios modelos de objetos implementados.

VREP será el encargado de publicar la información de posición, los puntos obtenidos por el LiDAR y las imágenes capturas por la cámara. Estos datos serán utilizados como punto de partida para poder realizar la fusión sensorial.

3.3. Calibración y fusión sensorial

Para poder implementar el algoritmo de detección y seguimiento de objetos planteado es necesario realizar previamente una calibración de los sensores y la fusión de sus datos.

La fusión sensorial permite combinar la información de los puntos 3D obtenidos por el láser junto con el color correspondiente de cada uno de esos puntos en función de la imagen capturada por la cámara.

Para realizar esta etapa se hará uso del conjunto de paquetes *but_velodyne* que se encuentran disponibles desde el repositorio Github [19]. De este repositorio se obtienen 3 paquetes:

- but_velody_proc: se utiliza para el procesamiento de datos del láser 3D.
- but_calibration_camera_velodyne: contiene las herramientas para la calibración de la cámara con el LiDAR 3D.
- but_velodyne_odom: permite estimar la odometría en función de las nubes de puntos y el color.

En este trabajo se empleará el paquete *but_calibration_camera_velodyne* que incluye nodos de ROS para poder realizar la estimación de los 6 parámetros de calibración (3 de posición y 3 de orientación).

El método empleado para la calibración [20] permite estimar tanto la posición como la orientación (6 grados de libertad) de la cámara y el LiDAR en dos etapas (una gruesa y otra fina) empleando un marcador 3D que facilita que los sensores detecten de forma robusta los bordes en una sola observación.

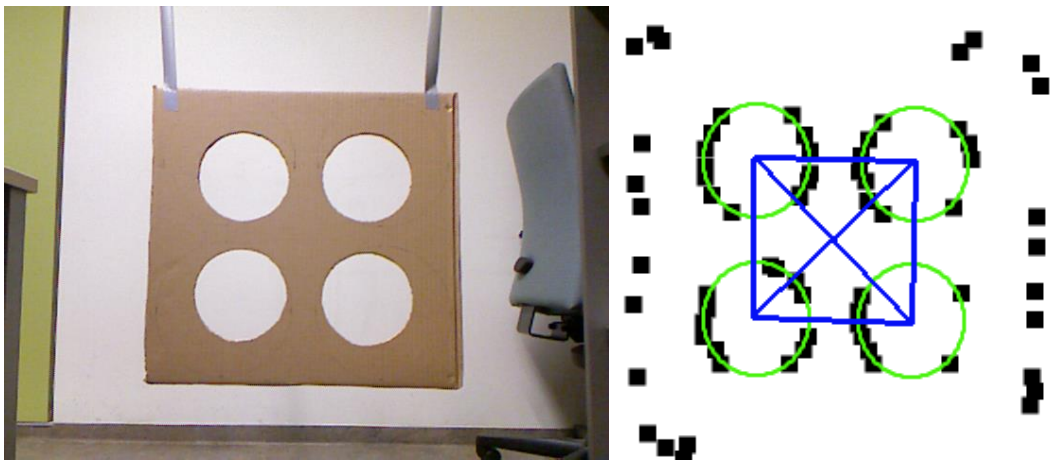

FIGURA 3.3-1: Marcador 3D empleado para la calibración en [20].

En la primera etapa (calibración gruesa) se da prioridad a los tres parámetros de traslación sobre los de rotación empleando el marcador 3D para encontrar las correspondencias mediante la transformada de Hough y eliminando los valores atípicos empleando un algoritmo basado en RANSAC; mientras que en la segunda etapa se refina la calibración incluyendo los parámetros de rotación realizando una búsqueda en un pequeño subespacio.

Este paquete de ROS incluye 3 nodos: dos de ellos para la calibración (gruesa y fina) y un tercero para la publicación de la nube de puntos empleando una calibración pre-calculada, que será el utilizado en este trabajo y que se denomina *coloring*.

3.3.1. Transformación de coordenadas

Para poder fusionar los datos obtenidos por el LiDAR con la imagen capturada por la cámara y obtener una nube de puntos a color es necesario saber dónde está cada sensor ubicado respecto a un marco de referencia común. Este problema se soluciona con la biblioteca de ROS *tf*, que gestiona el topic /tf que permite definir las transformaciones de posición entre diferentes marcos de coordenadas.

Podemos visualizar el árbol de transformadas del sistema empleando la herramienta rqt, que mostrará el siguiente esquema:

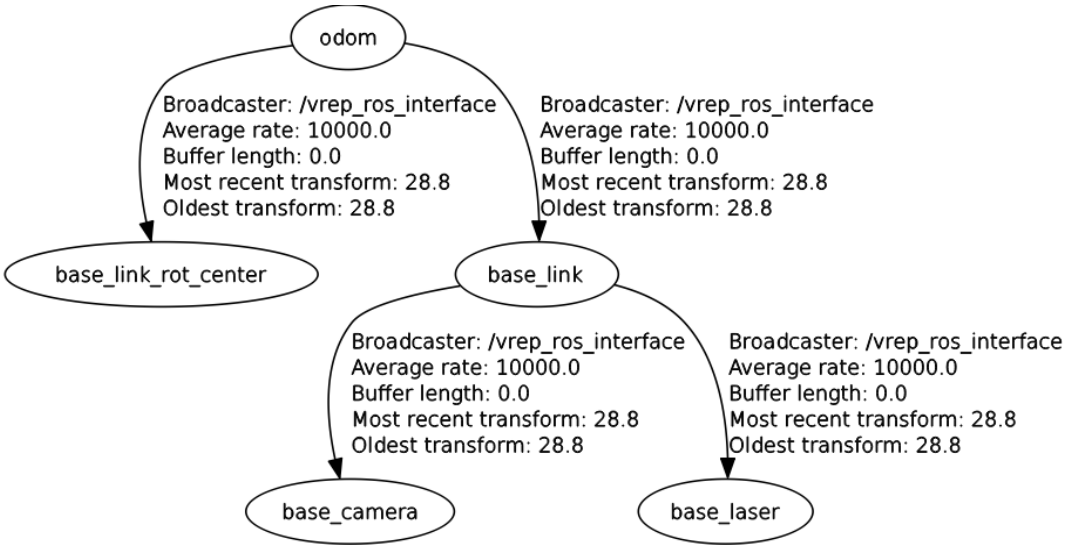

FIGURA 3.3-2: Árbol de transformadas del sistema de simulación.

En el grafo se puede ver el marco de coordenadas del coche sensorizado, donde el marco de referencia es *odom*, que se relaciona con *base_link_rot_center* (centro de rotación del vehículo) y *base_link*. A este último se encuentran referenciadas las coordenadas base de la cámara y del láser. Todas las relaciones entre los diferentes marcos de coordenadas son llevadas a cabo por el nodo */vrep_ros_interface*.

En la siguiente tabla e imágenes podemos apreciar cuál es la relación entre los marcos en valores numéricos de traslación y rotación, y su ubicación sobre el vehículo, que podemos visualizar en rviz:

Marco de coordenadas	**Posición Absoluta**			**Orientación Absoluta**			
	X	**Y**	**Z**	**R**	**P**	**Y**	**w**
odom	0	0	0	1.92e-9	-5.69e-9	5.96e-8	1
base_link_rot_center	-1.2	0	-0.55	-1.92e-9	5.69e-9	-5.96e-8	1
base_link	0	0	0	-1.92e-9	5.69e-9	-5.96e-8	1
base_camera	0	0	1.1	-0.5	0.5	-0.5	0.5
base_laser	0	0	1.002	0.707	2.02e-6	2.06e-6	0.707

TABLA 3.3-1: Posición y orientación de los marcos de coordenadas en simulación.

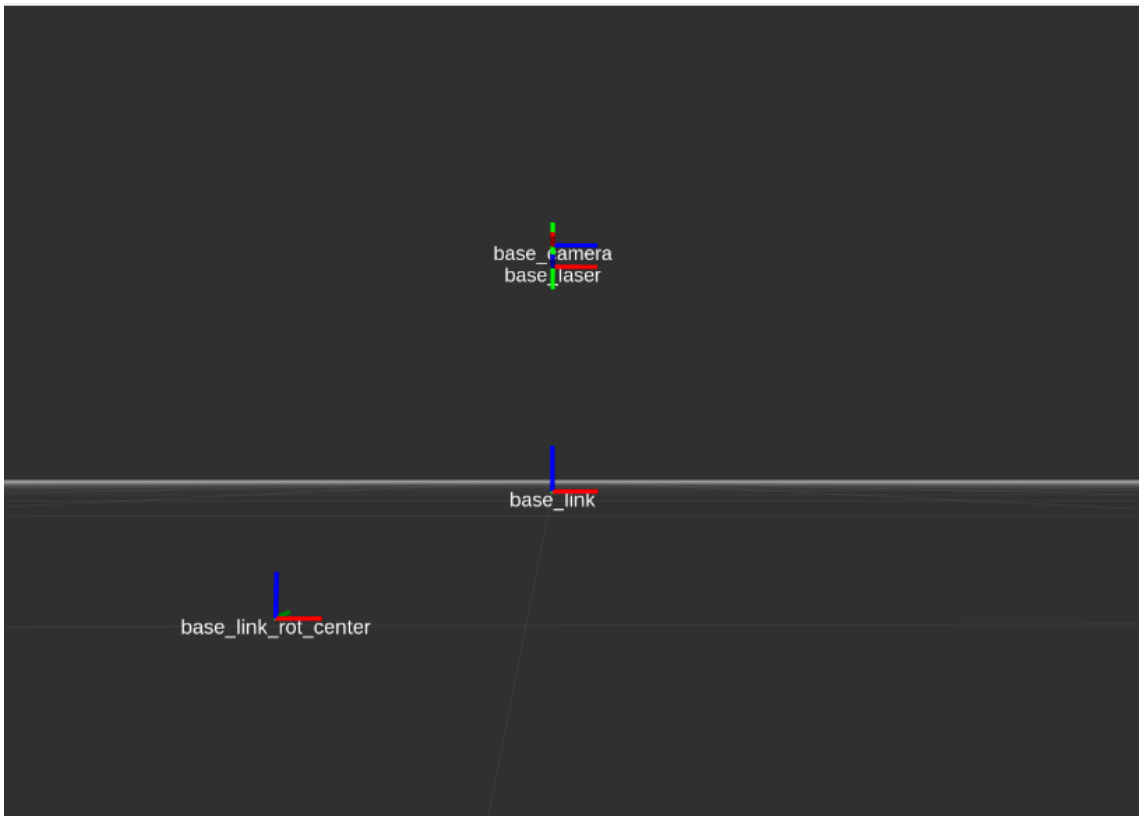

FIGURA 3.3-3: Marcos de coordenadas de simulación en rviz (sin modelo).

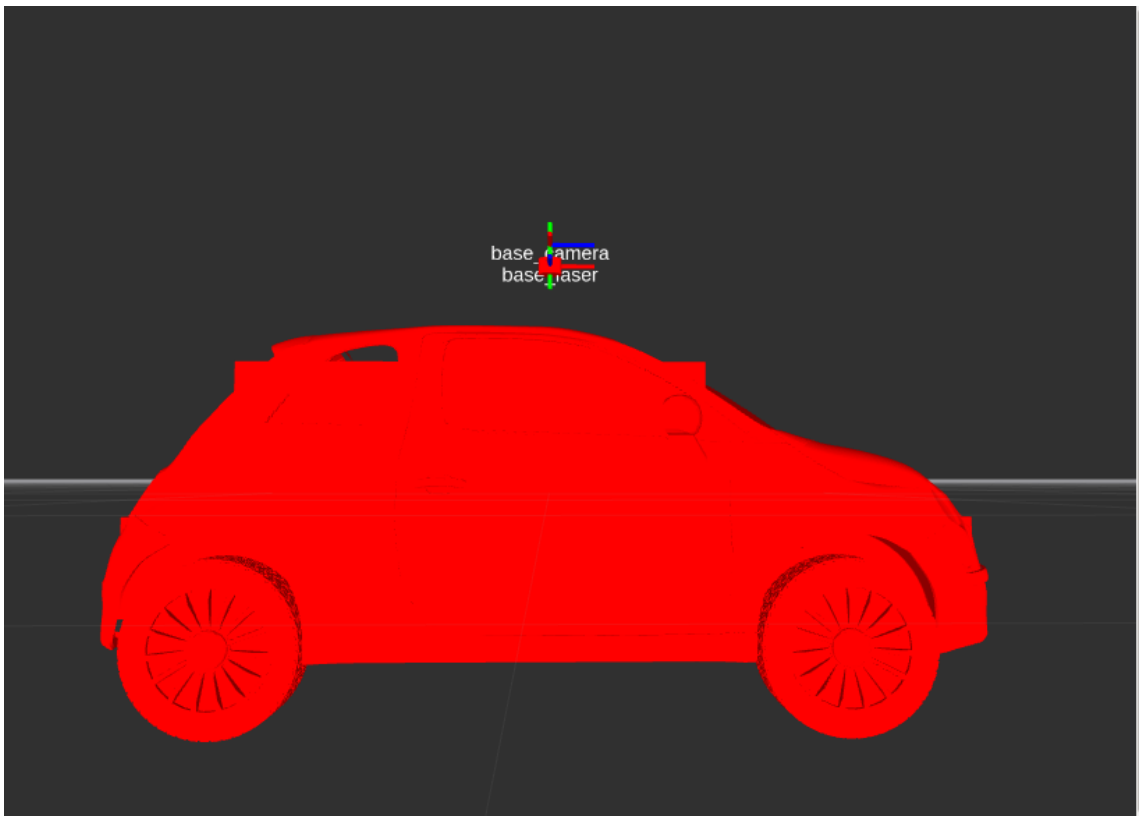

FIGURA 3.3-4: Marcos de coordenadas de simulación en rviz (con modelo).

El launch *navigation.launch* además de lanzar rviz, cargar el modelo del vehículo y ajustar los parámetros, llama al nodo *local_transform_velodyne* que se encarga de transformar la nube de puntos publicada por el LiDAR en el marco de coordenadas global, al marco de coordenadas local del láser. Para ello debe suscribe al topic */velodyne_points* y publicar el topic */velodyne_points_local*, ambos con mensajes del tipo *sensor_msgs/PointCloud2*.

El nodo se mantiene a la espera de una transformada del marco de coordenadas de */odom* a */baser_laser* para aplicarla al mensaje de la nube de puntos contenida en /velodyne_points y publicarlo como /velodyne_points_local:

```
sensor_msgs::PointCloud2 local_PC; //transformed point cloud

/* This tries to transform the point cloud according to the latest
available Transform */

tf::StampedTransform StmpdTransform;

tf::Transform net_transform(StmpdTransform.getRotation(),
                            StmpdTransform.getOrigin());

net_transform=net_transform.inverse();
pcl_ros::transformPointCloud("/base_laser",net_transform,
                             *msg, local_PC);

/*This tries to transform the point cloud according to the next
available Transform*/

listener.waitForTransform("/odom","/base_laser",ros::Time::now(),
                          ros::Duration(1.0)); //wait for tf update

pcl_ros::transformPointCloud( "/base_laser", *msg, local_PC,
                              listener); //"/base_laser" frame

pub.publish(local_PC);
```

3.3.2. Configuración e implementación

Una vez que los puntos de láser están en el marco de coordenadas local se hará uso del launch *coloring.launch* para fusionarlos con los datos de la cámara. Este launch va a cargar los parámetros de calibración *calibration_vlp16.yaml* y *coloring_vlp16.yaml*, que utilizará el nodo *coloring-node*.

El nodo se suscribirá al topic de los puntos del láser transformado a coordenadas locales (*/velodyne_points_local)* y a los topics de la cámara (*camera/left/image/raw* y */camera/left/camera_info*) que transportan mensajes de la imagen capturada y de información de la cámara, respectivamente; y publicará el topic */velodyne_coloured_points* que contiene la información de la nube de puntos a color.

El archivo de configuración *coloring.yaml* contiene los parámetros de calibración calculados por el nodo de calibración del paquete *but_velodyne*, y que ya ha sido calibrado previamente. Este archivo especifica dos tipos de parámetros:

- 6DoF: vector de parámetros de calibración calculados por el proceso de calibración.
- velodyne_color_topic: nombre del topic donde se publicará la nube de color coloreada.

En nuestro caso este archivo contendrá la siguiente información:

```
but_calibration_camera_velodyne:

   6DoF: [0.00, 0.098, 0.0, -1.57, -1.57, -3.14]
   velodyne_color_topic: /velodyne_coloured_points
```

El archivo de configuración *calibration.yaml* contiene los parámetros de la cámara y del láser. Este archivo especifica 6 tipos de parámetros:

- camera_frame_topic: nombre del topic donde se publica la imagen RGB de la cámara.
- camera_info_topic: topic con la información de la cámara del tipo CameraInfo.
- velodyne_topic: topic donde se publica la nube de puntos del LiDAR.
- marker: tamaño físico del marcador 3D para la calibración.
- circles_distance: distancia entre los centros del círculo del marcador.
- circles_radius: radio de los círculos del marcador.

En nuestro caso este archivo contendrá la siguiente información:

```
but_calibration_camera_velodyne:

  camera_frame_topic: /camera/left/image_raw
  camera_info_topic: /camera/left/camera_info
  marker: {circles_distance: 0.23, circles_radius: 0.0825}
  velodyne_topic: /velodyne_points_local
```

El nodo coloring lee la nube del puntos del topic /velodyne_points_local y le aplica la transformada correspondiente según los parámetros de calibración configurados en el fichero *coloring.yaml*.

Para cada punto 3D del LiDAR se calcula su proyección 2D utilizando la matriz de proyección obtenida del topic */CameraInfo*:

$$\begin{bmatrix} wx \\ wy \\ w \end{bmatrix} = M_{3x4} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

```
void cameraInfoCallback(const sensor_msgs::CameraInfoConstPtr& msg)
{
 float p[12];
 float *pp = p;

 for (boost::array<double, 12ul>::const_iterator i = msg->P.begin();
      i != msg->P.end(); i++)
 {
       *pp = (float)(*i);
       pp++;
 }
  cv::Mat(3, 4, CV_32FC1, &p).copyTo(projection_matrix);
}
```

De este modo se asigna a cada punto del láser el color correspondiente del píxel de la imagen que se obtiene del topic */image_raw*, teniendo en cuenta el número de haces del LiDAR y la transformación 2D-3D.

Debe considerarse que el paquete por defecto está pensado para trabajar con un Velodyne LiDAR HDL-32, mientras que en este trabajo se va a hacer uso del Velodyne LiDAR VLP-16, por lo que en el fichero cabecera *Velodyne.h* se modificará el parámetro *RINGS_COUNT* de 32 a 16.

Por último, se aplica una rotación a la nube de puntos a color para que coincida con el marco de coordenadas de visualización de rviz, obteniendo el siguiente resultado:

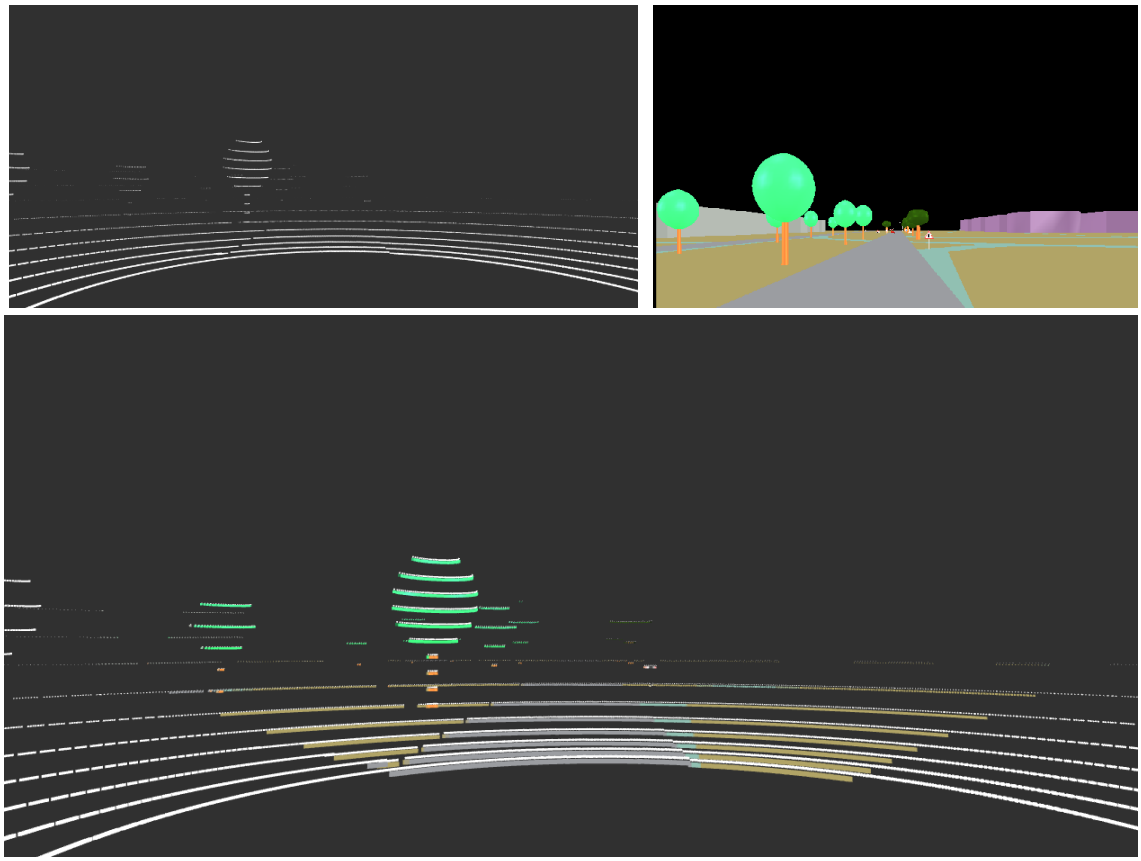

FIGURA 3.3-5: Proceso de fusión sensorial.

3.3.3. Sincronización de mensajes

Para poder obtener la odometría del coche (*/odom*) que corresponde en cada instante a la nube de puntos coloreada adquirida de la fusión sensorial (*/velodyne_coloured_points*), es necesario que exista una sincronización entre ambos topics. Para ello se empleará la biblioteca de ROS *message_filters* [21] que contiene algunos algoritmos de filtrado de mensajes entre los que se encuentran la sincronización del tiempo entre diferentes mensajes.

El filtro de sincronización se basa en una política que determina como sincronizar los distintos canales. Existen dos políticas: ExactTime y ApproximateTime. En nuestro caso emplearemos la primera de ellas que requiere que los mensajes tengan exactamente la misma marca de tiempo.

En el siguiente extracto de código puede apreciarse su utilización:

```
ros::NodeHandle nh;
message_filters::Subscriber<sensor_msgs::PointCloud2> lidar_sub_(nh,
                                                    "/velodyne_coloured_points", 1);
message_filters::Subscriber<nav_msgs::Odometry> odom_sub_(nh, "/odom", 1);

typedef sync_policies::ApproximateTime<sensor_msgs::PointCloud2, nav_msgs::Odometry>
                                                                        MySyncPolicy;

Synchronizer<MySyncPolicy> sync(MySyncPolicy(10), lidar_sub_, odom_sub_);
sync.registerCallback(boost::bind(&callback, _1, _2)); //2 topics
```

Capítulo IV

DETECCIÓN Y SEGUIMIENTO DE OBJETOS

En este capítulo se explicarán las implementaciones llevadas a cabo en la detección y seguimiento de objetos para cumplir con los objetivos planteados.

4.1. Detección

El problema de detección puede definirse como el proceso de reconocimiento de un objeto y la determinación de su información de identidad y estado. Como ya se ha mencionado, en este trabajo se emplea el enfoque de seguimiento por detección (*tracking-by-detection*), por lo que el proceso de seguimiento toma como datos de partida los resultados del proceso de detección para así poder conocer el todo momento el estado actual y la evolución de estados de los objetos.

La detección de objetos con el sistema formado por el LiDAR 3D y la cámara se basa en un proceso de filtrado, segmentación y clusterizado, que permite determinar las nubes de puntos de los objetos que luego serán empleadas para el seguimiento. En cada paso de tiempo en el proceso de detección se realizarán las etapas de forma secuencial, como se muestra en el siguiente esquema:

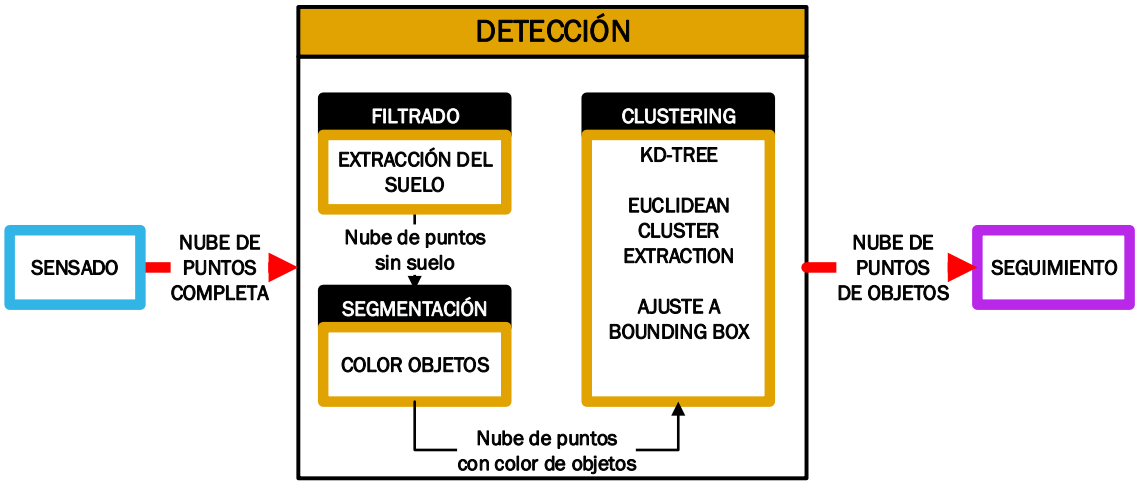

FIGURA 4.1-1: Esquema de detección de objetos.

Cuando se trabaja con una gran cantidad de datos de nubes de puntos se requiere una potencia computacional elevada para realizar su procesamiento, además, como los puntos son elementos discretos que representan una escena real continua, resulta eficiente combinarlos para formar grupos de puntos con características comunes.

Esto implica que los datos brutos obtenidos por los sensores deben procesarse previamente para poder eliminar los puntos que resultan innecesarios y así reducir la complejidad de la escena antes de detectar los objetos de forma individual. El proceso de segmentación permite diferenciar entre los objetos que no son de interés para ser detectados, como el suelo o los árboles, que forman parte del entorno, y los objetos de interés, como los automóviles y los peatones.

En este trabajo se va a hacer uso de la segmentación basada en los objetos que suele ser habitual en los esquemas de *tracking-by-detection*[4]. Este enfoque se basa en emplear un modelo de puntos para describir los objetos, que requiere posteriormente de una estimación de la posición y un filtro de seguimiento que permita identificar el objeto segmentado en diferentes instantes de tiempo.

El proceso de detección se realiza en varias etapas: una extracción del suelo para separar los objetos que deben detectarse, de la zona de la escena que no es de interés y así reducir la dimensión de los datos sin procesar; una segmentación por color de la nube de puntos que pertenece a un objeto; y una agrupación de puntos para formar objetos (*clustering*) según su forma y color, y ajuste a un cubo que delimita y uniformiza sus dimensiones.

[4] Los esquemas de seguimiento que presentan una estructura de seguimiento antes de detección (*detection-by-tracking*) suelen emplear un enfoque de segmentación basado en rejillas de ocupación, aunque ambos métodos pueden mezclarse.

4.1.1. Estructura de datos de la nube de puntos

Lo primero que debe hacerse para poder trabajar con la nube de puntos es leer el mensaje que contiene el topic */velodyne_coloured_points* publicado por la fusión sensorial en el apartado anterior. El mensaje que contiene es del tipo *sensor_msgs::PointCloud2*.

Este mensaje contiene un conjunto de puntos de n-dimensiones que pueden incluir información adicional como normales o intensidad. Los datos de los puntos se almacenan como un *blob* binario y su diseño se describe mediante el contenido de la matriz *fields*. Los datos de la nube de puntos pueden organizarse en 2D (como una imagen) o en 1D (desordenados). En la siguiente tabla podemos ver los campos que contiene este mensaje:

Campo	Descripción
std_msgs/Header header	Metadatos estándar de alto nivel que contiene información de marcas de tiempo de un determinado marco de coordenadas.
uint32 height	Dato de altura de la nube de puntos para su representación 2D. Si la nube se describe en 1D este campo se pone a 1.
uint32 width	Dato de ancho de la nube de puntos para su representación 2D. Si la nube se describe en 1D este campo describe la longitud de la nube.
sensor_msgs/PointField[] fields	Contiene la descripción de los puntos en el formato PointCloud2.
bool is_bigendian	Determina si los datos están en formato big endian (orden en memoria).
uint32 point_step	Indica la longitud de un punto en bytes.
uint32 row_step	Indica la longitud de una fila en bytes.
uint8[] data	Indica el tamaño de memoria del dato actual (row_step*height).
bool is_dense	Es verdadero si no existen puntos no válidos.

TABLA 4.1-1: Campos del mensaje sensor_msgs::PointCloud2.

Una vez definidos sus campos, es necesario convertir el mensaje en una nube de puntos para poder trabajar con él, para lo cual se emplean las siguientes funciones:

```
pcl_conversions::toPCL(*msg,*Input_Cloud);
pcl::fromPCLPointCloud2(*Input_Cloud, *vlp_cloud_Ptr);
```

4.1.2. Filtrado: Extracción del suelo

La nube de puntos que procede de la fusión sensorial del LiDAR 3D y la cámara obtenida en el apartado anterior incluye información del suelo que no es detectada como un obstáculo, ya que tanto los vehículos como los peatones se desplazan por encima de él. Resulta útil determinar la ubicación del suelo para establecer la cota mínima de altura sobre la que van a situarse los objetos de interés.

La extracción del suelo de la nube de puntos es un proceso preliminar esencial en la detección de objetos, ya que un porcentaje elevado de los puntos aportados por el láser van a pertenecer al suelo, y al no tratarse de información útil para el seguimiento, su eliminación reduce la carga computacional en la detección de los objetos.

El método empleado en este trabajo para la extracción del suelo se basa en obtener una superficie plana en función de los puntos escaneados con menor altura, para después aplicar RANSAC y así eliminar los outliers y quedarse con los puntos que geométricamente pertenecen a la ecuación del plano calculada.

Al considerar la simplificación de que todo el suelo es un único plano se consiguen evitar las posibles inconsistencias de diferentes elevaciones entre puntos vecinos que pueden resultar al analizar el terreno punto a punto, o por zonas, además de que evita problemas de oclusiones a la hora de detectar el suelo ya que considera que todos los puntos que pertenecen al plano forman parte del suelo.

El método empleado para segmentar el suelo se basa en el algoritmo de segmentación planar que se incluye en la documentación de PCL. A partir de la nube de puntos con la que se va a trabajar, lo primero que debe hacerse es crear un objeto *SACSegmentation* y configurar el modelo y tipo de método empleado. Se definirá como tipo de dato la nube de puntos con color (*pcl::PointXYZRGB*), el modelo empleado será el que permite extraer el plano (*SACMODEL_PLANE*) y el método utilizado será RANSAC (*SAC_RANSAC*).

El método de RANSAC (*RANdom SAmple Consensus*) se caracteriza por ser uno de los más sencillos y empleados, gracias a sus buenos resultados, por lo que suele utilizarse como base para otros métodos más complejos. Se trata de un algoritmo iterativo que permite calcular los parámetros de un modelo matemático (en este caso un plano) de un conjunto de datos que va a dividir en dos tipos: *inliers* (o datos que se ajustan al modelo) y *outliers* (o datos que no se ajustan al modelo). El resultado va a estar sujeto a un intervalo de confianza y al número de iteraciones que realice. El funcionamiento de este método puede resumirse en los siguientes pasos:

1. Selecciona un subconjunto de datos del total de forma aleatoria y genera un modelo que se ajusta a ellos.

2. Se compara el resto de los datos con el modelo, y los que se ajustan con un determinado intervalo de desviación, se consideran que forman parte del conjunto de consenso.

3. Si existe un número suficiente de datos que pertenecen al conjunto de consenso, se considera que el modelo estimado es bueno. El modelo puede reajustarse volviendo a establecer un modelo únicamente con los datos del conjunto.

4. En caso de no validarse el modelo, se volvería al primer punto considerando otro subconjunto aleatorio de datos.

Este método termina, bien porque ha convergido a una solución aceptable dentro del intervalo definido, o porque ha agotado el número de iteraciones establecidas. Podemos ver un ejemplo visual sencillo de ajuste al modelo de una recta en la siguiente imagen:

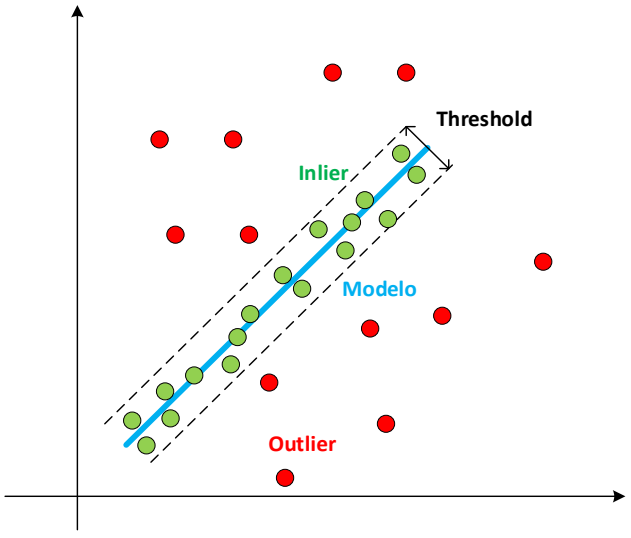

FIGURA 4.1-2: Ejemplo de ajuste a una recta mediante RANSAC.

Los parámetros de máximo número de iteraciones (100) y umbral (0,15), que determina cuando un dato se ajusta al modelo, se imponen en base a una evaluación experimental que permite obtener los mejores resultados. Podemos ver la configuración de estos parámetros en el siguiente extracto de código:

```
// Create the segmentation object for the planar model and set all the parameters
        pcl::SACSegmentation<pcl::PointXYZRGB> seg;
        pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
        pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);

        seg.setOptimizeCoefficients (true);
        seg.setModelType (pcl::SACMODEL_PLANE);
        seg.setMethodType (pcl::SAC_RANSAC);
        seg.setMaxIterations (100);
        seg.setDistanceThreshold (0.15);
```

Como resultado la segmentación nos dará los coeficientes (A, B, C, D) del plano que contiene una mayor cantidad de puntos, que presumiblemente corresponderán al suelo, y que viene dado por la siguiente ecuación:

$$A \cdot x + B \cdot y + C \cdot z + D = 0$$

Una vez que la nube de puntos se encuentra segmentada en inliers (plano del suelo) y outliers (el resto de la nube) podemos extraer sus índices para eliminar los puntos correspondientes al suelo. De este modo obtenemos el siguiente resultado:

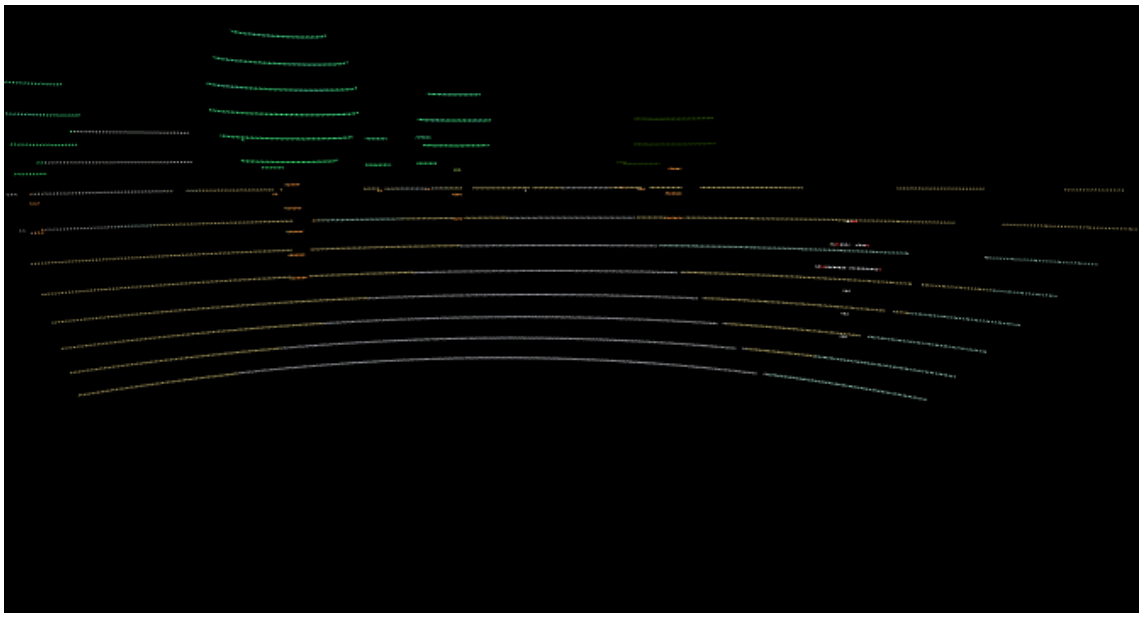

FIGURA 4.1-3: Nube de puntos completa a color.

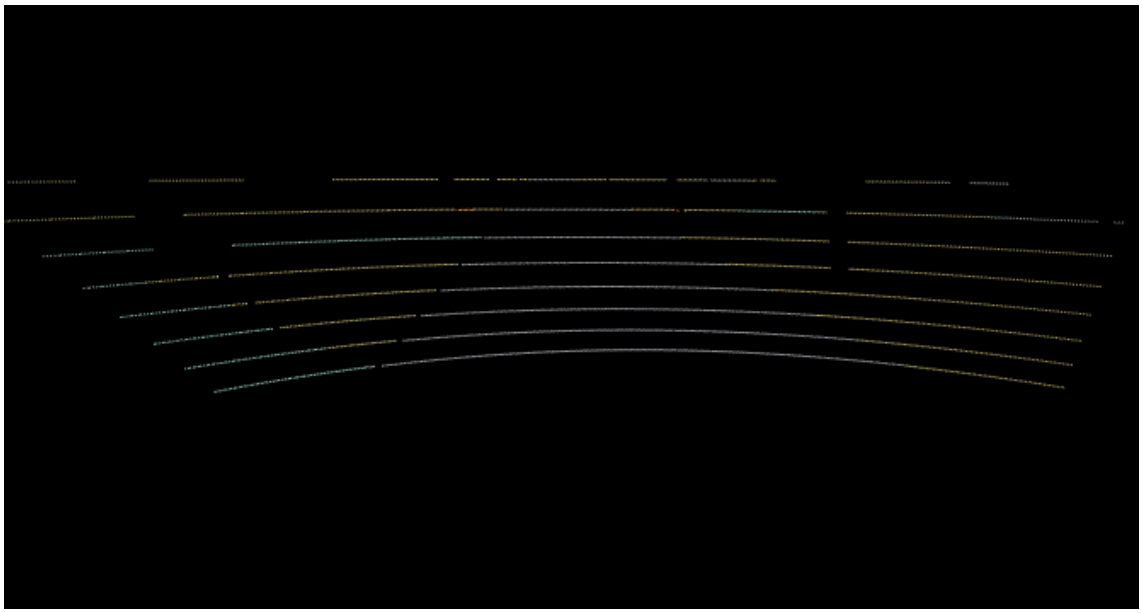

FIGURA 4.1-4: Extracción del suelo de la nube de puntos completa a color.

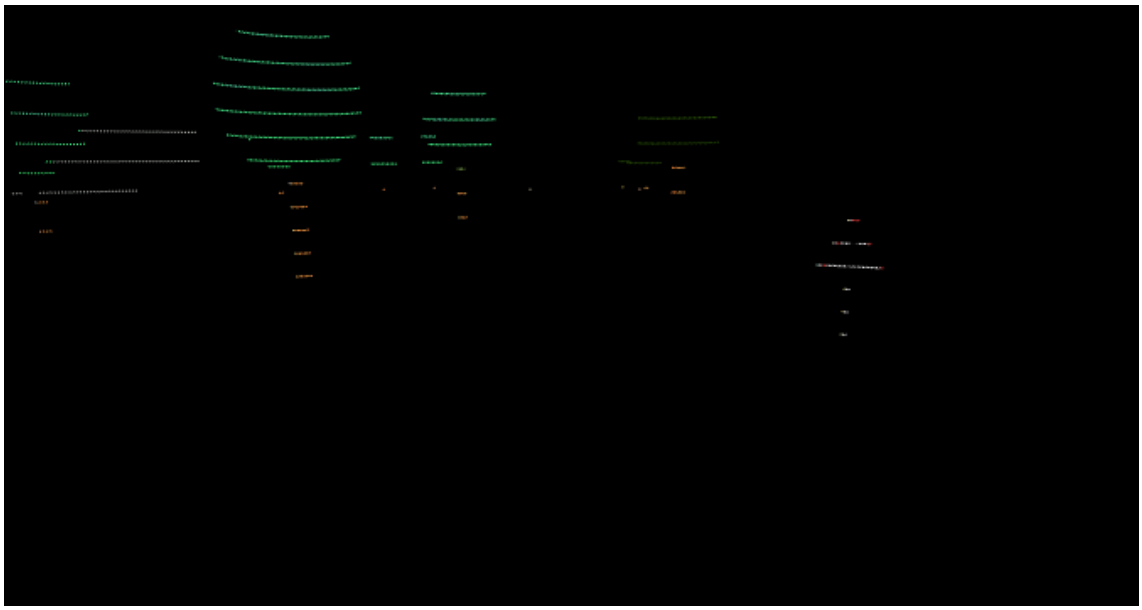

FIGURA 4.1-5: Nube de puntos a color sin suelo.

4.1.3.Segmentación de objetos

Antes de dividir la nube de puntos en los diferentes clústeres que pueden componer la escena, debe considerarse que no todos ellos serán útiles poder realizar el seguimiento, ya que en la escena encontraremos objetos como árboles, señales o edificios.

Como en este trabajo se centrarán los esfuerzos de seguimiento únicamente en vehículos y peatones, será necesario emplear un método que permita dividirlos del resto, para lo cual se tomará como base la idea de la segmentación semántica, gracias a la información de color disponible de la cámara.

El problema de la segmentación semántica pertenece al ámbito de la visión por computador y se basa en etiquetar cada píxel de una imagen, delimitando los objetos por su forma real. Se trata de una tarea desafiante en la percepción de vehículos inteligentes, para la que se emplean redes neuronales profundas (DNNs) que permiten la clasificación de la imagen en múltiples categorías.

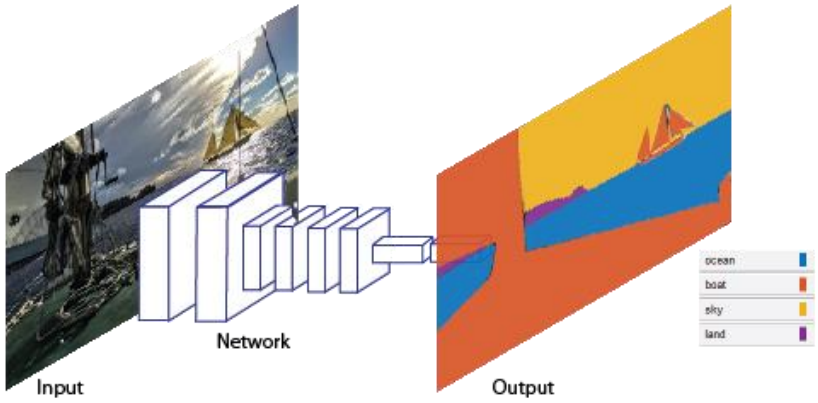

FIGURA 4.1-6: Ejemplo esquemático de segmentación semántica mediante una red neuronal.

Partiendo de esta base en la simulación mediante VREP, en lugar de emplear un algoritmo de clasificación mediante red neuronal, se etiquetarán manualmente los vehículos y peatones con un código de color único que permita clasificar los píxeles de la imagen:

Objeto	R	G	B
Coche	255	0	0
Peatón	156	184	237

TABLA 4.1-2: Colores de los objetos en VREP.

FIGURA 4.1-7: Modelos de vehículo y peatón en VREP, de izquierda a derecha.

Aprovechando esta característica de color se aplicará un filtro condicional (*ConditionalRemoval*) a la nube de puntos en función del objeto. Como condición (*ConditionAnd*) se establecerá un rango de valores máximos y mínimos de colores RGB en el que se asegura que se detectan los objetos. De este modo cada vez que se analice la nube de puntos todos aquellos que cumplan las condiciones de color se clasificarán según el tipo de objeto establecido.

```
pcl::ConditionAnd<pcl::PointXYZRGB>::Ptr color_cond (new pcl::ConditionAnd<pcl::PointXYZRGB> ());
color_cond->addComparison (pcl::PackedRGBComparison<pcl::PointXYZRGB>::Ptr (new
                    pcl::PackedRGBComparison<pcl::PointXYZRGB> ("r", pcl::ComparisonOps::LT, rMax)));
color_cond->addComparison (pcl::PackedRGBComparison<pcl::PointXYZRGB>::Ptr (new
                    pcl::PackedRGBComparison<pcl::PointXYZRGB> ("r", pcl::ComparisonOps::GT, rMin)));
color_cond->addComparison (pcl::PackedRGBComparison<pcl::PointXYZRGB>::Ptr (new
                    pcl::PackedRGBComparison<pcl::PointXYZRGB> ("g", pcl::ComparisonOps::LT, gMax)));
color_cond->addComparison (pcl::PackedRGBComparison<pcl::PointXYZRGB>::Ptr (new
                    pcl::PackedRGBComparison<pcl::PointXYZRGB> ("g", pcl::ComparisonOps::GT, gMin)));
color_cond->addComparison (pcl::PackedRGBComparison<pcl::PointXYZRGB>::Ptr (new
                    pcl::PackedRGBComparison<pcl::PointXYZRGB> ("b", pcl::ComparisonOps::LT, bMax)));
color_cond->addComparison (pcl::PackedRGBComparison<pcl::PointXYZRGB>::Ptr (new
                    pcl::PackedRGBComparison<pcl::PointXYZRGB> ("b", pcl::ComparisonOps::GT, bMin)));
// build the filter
pcl::ConditionalRemoval<pcl::PointXYZRGB> condrem (color_cond);
condrem.setInputCloud (cloud_filtered);
condrem.setKeepOrganized(false);

// apply filter
condrem.filter (*cloud_filtered);
```

Los rangos aplicados en función de cada objeto son los que se muestran en la siguiente tabla:

Objeto	Rmin	Rmax	Gmin	Gmax	Bmin	Bmax
Coche	125	255	0	55	0	0
Peatón	80	147	116	165	0	146

TABLA 4.1-3: Rangos de colores para el filtro de color de los objetos.

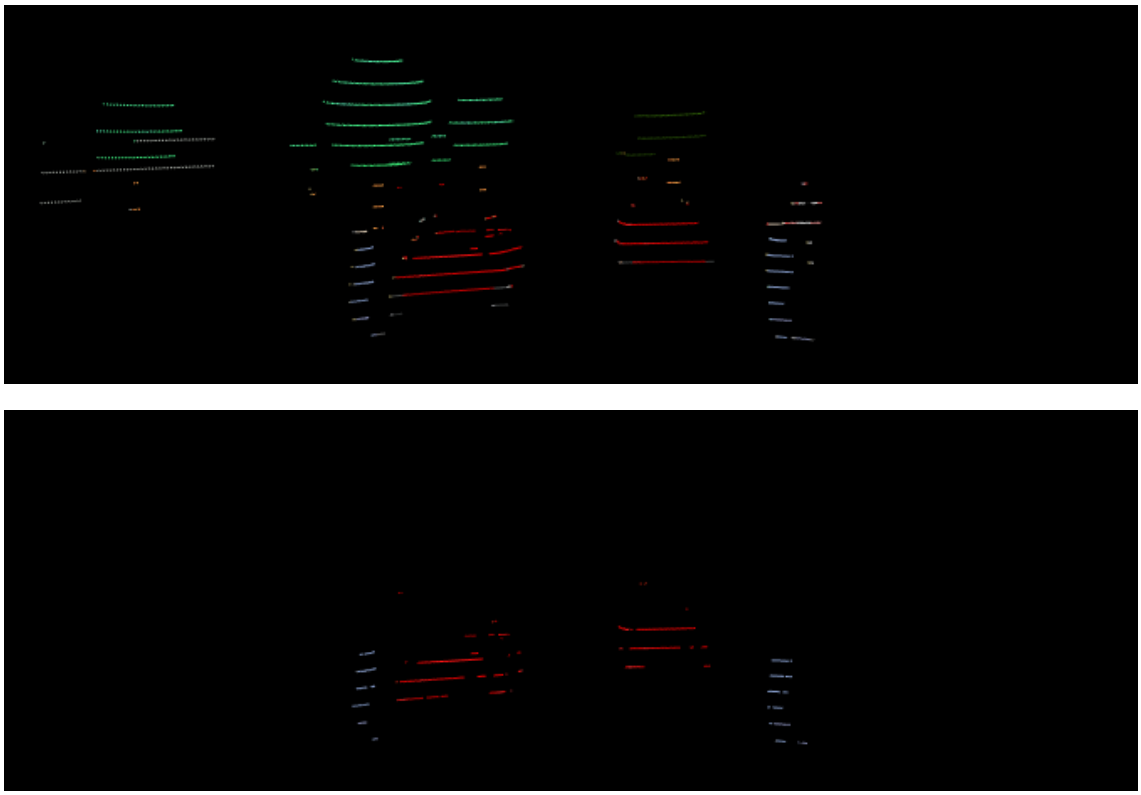

FIGURA 4.1-8: Comparativa de la imagen completa con el suelo extraído (arriba) y la imagen que detecta el color de los vehículos y peatones (abajo).

4.1.1.Clustering

Trabajar con toda la nube de puntos puede resultar costoso computacionalmente para realizar la detección y seguimiento sobre objetos individuales. Por este motivo se suele dividir la nube completa en nubes de puntos más pequeñas (*clusteres*) en la que cada una de ellas contiene puntos del espacio cercanos que pertenecen al mismo objeto.

En este trabajo se realiza un *clustering* basado en los datos 3D de la nube de puntos, que a pesar de requerir un mayor esfuerzo computacional que si se realizara con datos 2D, los resultados obtenidos son más precisos al incorporar la componente vertical de la altura. El método implementado se basa en la agrupación de puntos 3D mediante el vecino más cercano, para lo cual se emplea el algoritmo de búsqueda KD-Tree combinado con la extracción de clústeres en función de la distancia Euclídea (*Euclidean Cluster Extraction*), que se incluye en la documentación de PCL.

4.1.1.1. KD-Tree

KD-Tree (árbol de dimensión k) es una estructura de datos empleada para organizar puntos en un espacio euclídeo k-dimensional. Es un árbol de búsqueda binaria que sólo emplea planos perpendiculares a cada dimensión, y donde cada nodo contiene un punto, quedando todos los puntos atravesados por planos. Suele emplearse para las búsquedas de rango y del vecino más cercano. Como en este caso se va a trabajar con nubes de puntos de datos tridimensionales, se emplearán árboles 3D (k=3).

La organización en forma de árbol imita una estructura jerárquica dividida en niveles formados por nodos padre y nodos hijo, donde en cada nivel, los nodos se dividen mediante un plano perpendicular a un eje. La manera más eficiente para construirlo es empleando un ordenamiento rápido (o *Quicksort*) que se basa en tomar la mediana en una dimensión y ordenar el resto de los elementos a izquierda y derecha en función de si tienen un valor mayor o menor. Un ejemplo de proceso de estructuración de datos en forma de árbol se muestra en la siguiente imagen:

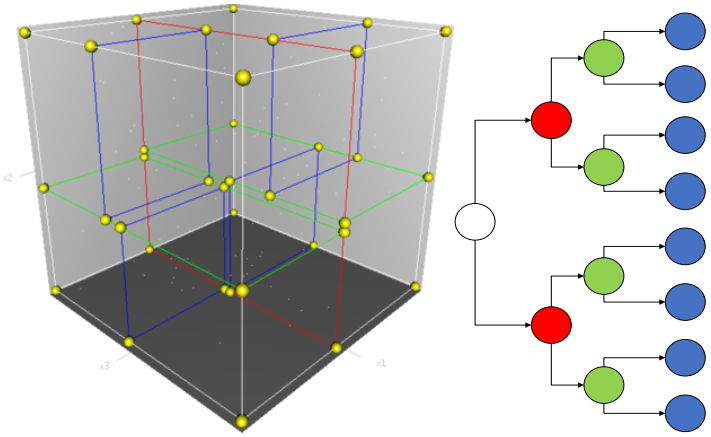

FIGURA 4.1-9: Ejemplo KD-Tree en 3D.

La celda raíz queda limitada por el cubo blanco. Se realiza la primera división (rojo) en una dimensión, dividiendo la raíz en dos subceldas. Cada una de estas subceldas es dividida (verde) por otra dimensión, resultando 4 subceldas, que a su vez se vuelven a dividir por la última dimensión (azul). En este caso con 8 subceldas quedan definidas las hojas del árbol, ya que se han cubierto todos los puntos, que representan los nodos del árbol. En caso de necesitar más subdivisiones, se volvería de nuevo a dividir por la primera dimensión.

4.1.1.2. Euclidean Cluster Extración

Como se ha comentado, el método de *clustering* se fundamentará en dividir la nube de puntos total en nubes más pequeñas según la distancia euclídea basada en el uso de los vecinos más cercanos según la división del espacio 3D realizada por el método KD-Tree explicado. El funcionamiento del algoritmo se expone a continuación:

1) Se crea un árbol KD-Tree para la nube de puntos.

```
pcl::search::KdTree<pcl::PointXYZRGB>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZRGB>);
tree->setInputCloud (cloud_filtered);
```

2) Se crea una lista de clústeres

```
std::vector<pcl::PointIndices> cluster_indices;
```

3) Para cada punto de la nube:

- **a)** Se busca el conjunto de puntos vecinos que pertenezcan a una esfera de radio inferior al umbral definido.
- **b)** Para cada punto vecino se comprueba si ya se ha evaluado, y si no, se añade.

```
pcl::EuclideanClusterExtraction<pcl::PointXYZRGB> ec;
ec.setClusterTolerance (tolerance);
ec.setMinClusterSize (min_cluster);
ec.setMaxClusterSize (max_cluster);
ec.setSearchMethod (tree);
ec.setInputCloud (cloud_filtered);
ec.extract (cluster_indices);
```

4) El algoritmo termina cuando todos los puntos forman parte de la lista de clústeres.

Para implementarlo se utilizará un objeto *EuclideanClusterExtraction* con el tipo de puntos *PointXYZRGB* de la librería PCL. El umbral se establecerá mediante el parámetro *setClusterTolerance* del objeto y se ha decidido que varíe en función del modelo de objeto detectado, del mismo modo que el tamaño mínimo y máximo de cada uno de los clústeres (parámetros *setMinClusterSize* y *setMaxClusterSize*, respectivamente).

Como es lógico, el número de puntos que se detectarán para un coche será superior al de un peatón, y además se tendrá en cuenta un umbral de análisis superior para los objetos de tipo vehículo para minimizar posibles problemas que pudiera causar la oclusión. De este modo se intentará evitar el problema de falsos positivos debido a un clúster muy pequeño o por el contrario demasiado grande y que pudiera agrupar varios objetos en uno. Con todo esto, los parámetros aplicados serán los que se presentan en la siguiente tabla:

Objeto	Umbral [m]	Cluster mínimo	Cluster máximo
Coche	1	25	400
Peatón	0.8	10	90

TABLA 4.1-4: Parámetros de clustering para cada uno de los objetos.

4.1.1.3. Ajuste a Boundingbox

Para poder realizar un seguimiento de los objetos es necesario conocer su posición, lo que incluye su dimensión y orientación. Conocer la posición de un objeto según un modelo de objeto establecido consiste en hacer coincidir el clúster con las dimensiones especificadas del modelo geométrico, como es en este caso una caja de contorno (*boundingbox*). Para cada uno de los objetos se consideran las siguientes dimensiones:

Objeto	Ancho [m]	Profundo [m]	Alto [m]
Coche	1.65	3.5	1.3
Peatón	0.4	0.4	1.8

TABLA 4.1-5: Dimensiones máximas del modelo de objetos.

Para poder ajustar la *boundingbox* al clúster, se calcula el centroide como el punto medio en cada una de las coordenadas del clúster detectado:

$$\text{centroide (XYZ)} = \frac{\sum \text{puntos (XYZ)}}{\text{número de puntos total}}$$

A partir de los valores máximos y mínimos de cada coordenada se calculan las dimensiones del clúster:

$$\text{ancho (w)} = x_{max} - x_{min}$$

$$\text{alto (h)} = y_{max} - y_{min}$$

$$\text{profundo (d)} = z_{max} - z_{min}$$

Tomando como referencia el centro de la nube de puntos detectada, se busca ajustar la *boundingbox*. Para el caso de la altura, se tomará como punto mínimo el plano del suelo extraído, a partir del cual se añadirá la altura determinada del objeto.

Para simplificar el proceso de detección de la orientación de los objetos, se considerarán objetos en la misma dirección de desplazamiento y se ajustará la *boundingbox* al tamaño establecido en la extracción de clústeres, teniendo en cuenta que en cualquier caso las dimensiones mínimas y máximas establecidas en el modelo de objeto.

En las siguientes imágenes puede apreciarse el proceso de ajuste de la boundingbox en una escena desde distintos puntos de vista:

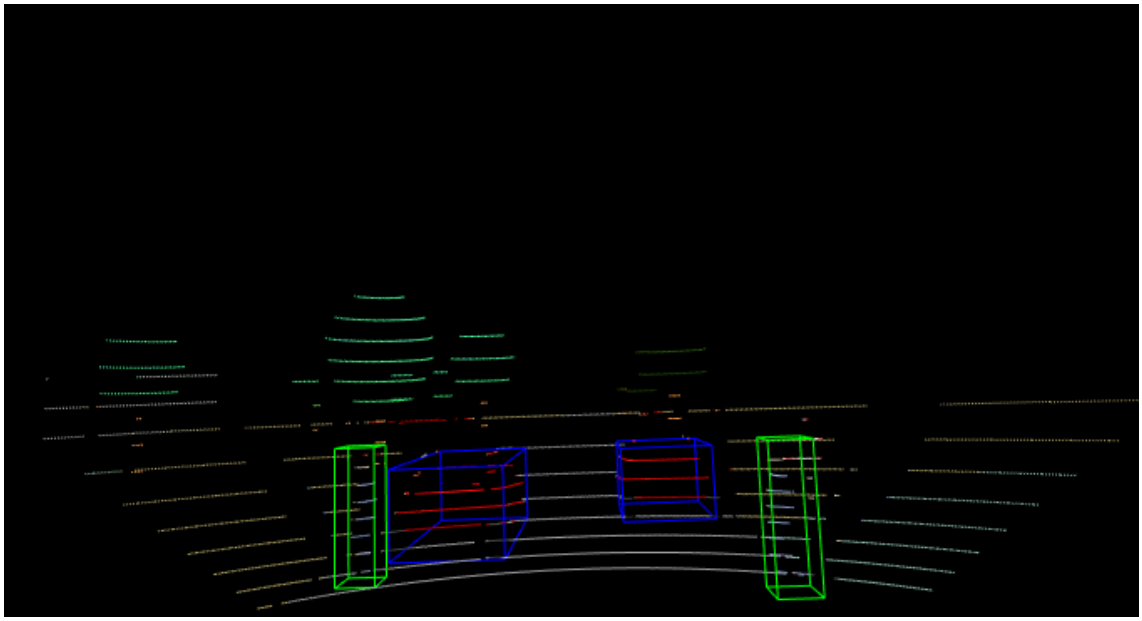

FIGURA 4.1-10: Ajuste de boundingbox azul a vehículos y verde a peatones (punto de vista conductor)

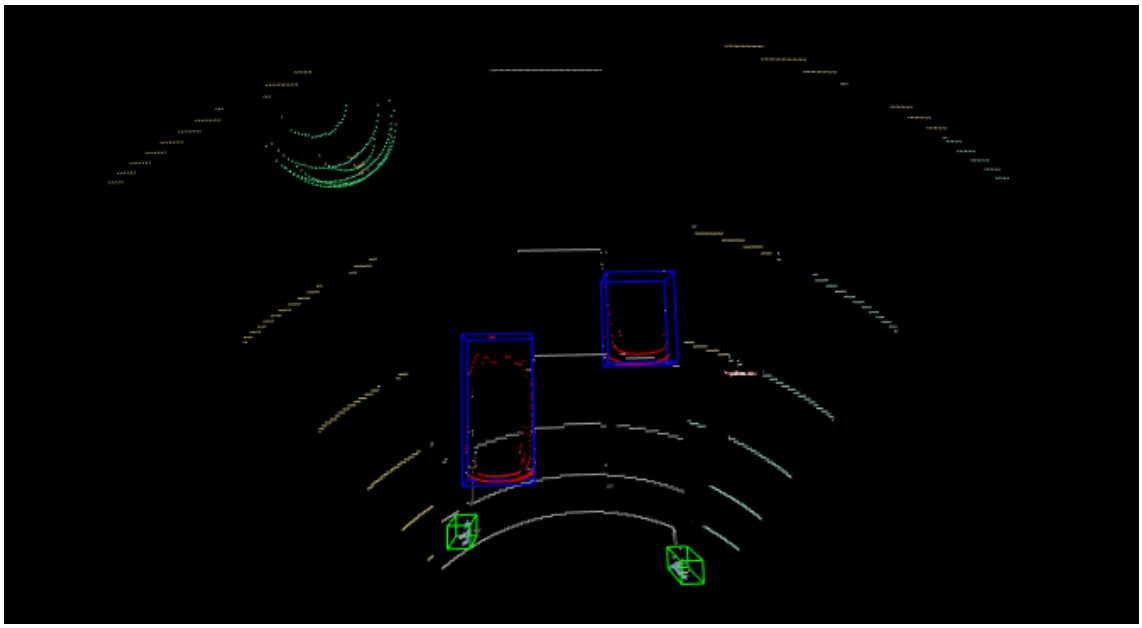

FIGURA 4.1-11: Ajuste de boundingbox azul a vehículos y verde a peatones (vista cenital)

4.2. Seguimiento

El seguimiento de objetos se entiende como el proceso que emplea las medidas de los sensores para determinar la posición, el movimiento y las características de los objetos, buscando su clasificación e identificación única.

En el contexto de la conducción autónoma el seguimiento de objetos es una tarea esencial, ya que el vehículo necesita conocer la posición y la velocidad de los objetos que lo rodean, para así poder ejecutar las acciones necesarias de control. El seguimiento de objetos permite predecir el desplazamiento de los objetos en movimiento que rodean al vehículo.

Para aclarar el proceso que se realiza en el seguimiento de objetos resulta útil el esquema que se muestra a continuación:

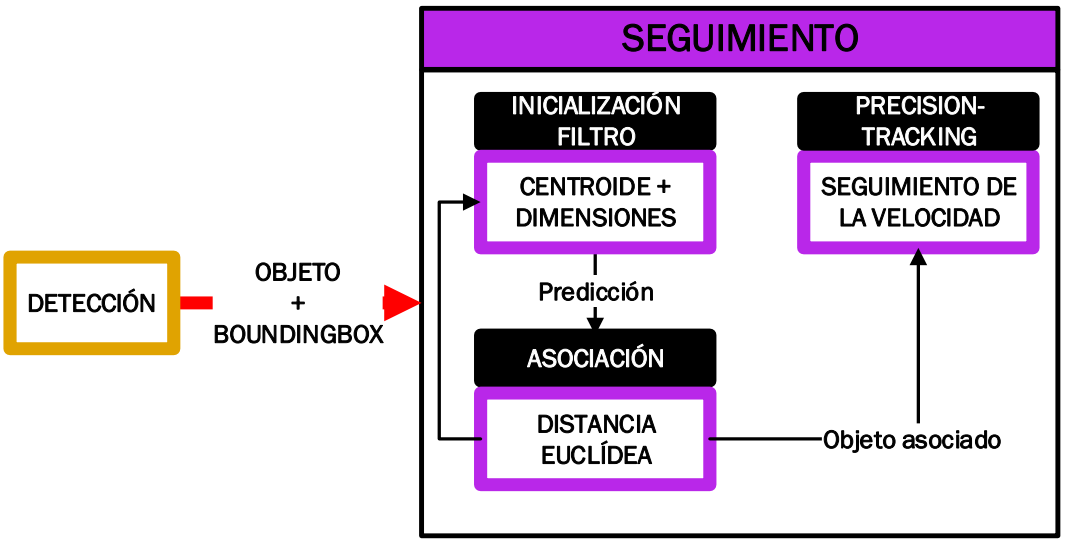

FIGURA 4.2-1: Esquema de seguimiento.

La salida del detector es una *boundingbox* con la dimensión y centroide, que se empleará para poder identificar cada objeto en cada instante de tiempo. Así, el resultado de la detección se utiliza para iniciar un nuevo filtro de seguimiento, o si ya existe uno, la medición verifica si puede tratarse del mismo mediante una asociación de datos, a partir de la estimación y predicción del estado actual.

En el problema de seguimiento, tanto las medidas de los sensores como la posición de los objetos seguidos se modelan como variables aleatorias, lo que implica que para poder calcular su evolución es necesario utilizar un enfoque probabilístico.

Para resolver el problema de seguimiento se emplea un filtro de Bayes (Kalman), que es un enfoque probabilístico general para estimar recursivamente una función de densidad de probabilidad desconocida utilizando mediciones y un modelo de proceso matemático.

Una vez ha sido asociado el objeto se pasan sus parámetros (nube de puntos y centroide) para generar un *tracker* empleando el algoritmo de precision-tracking que permite obtener la diferencia de velocidad entre dos instantes consecutivos de forma precisa.

4.2.1.Filtro de Kalman

Cuando las variables del filtro de Bayes son lineales y siguen una distribución normal, hablamos de un filtro de Kalman (KF). El filtro de Kalman permite identificar un estado no medible de un sistema dinámico lineal cuando está sometido a un ruido blanco aditivo. Si se conocen las varianzas de los ruidos que afectan al sistema, el filtro escoge de forma óptima la ganancia de realimentación del error. Además, al tratarse de un filtro recursivo, puede emplearse en tiempo real a partir de las medidas actuales, el estado anterior y la matriz de incertidumbre.

Por tanto, el objetivo de emplear el filtro de Kalman es poder estimar un estado (s(k)) de un proceso lineal a partir de un conjunto de medidas (z(k)). El filtro se utilizará para observar y predecir tanto la evolución de la posición de los objetos, como su dimensión, esperando que la posición evolucione según el modelo de movimiento, mientras que las dimensiones deberían mantenerse relativamente constantes. Como ambas evoluciones se encuentran alteradas por el ruido e incertidumbres, se realiza también un seguimiento de estas variables para aportar mayor coherencia a los resultados obtenidos.

En este trabajo al tratarse de un sistema en tiempo discreto, las funciones de estado (sin control) y medida que lo definen son las siguientes:

$$\mathrm{s(k)}_{12x1} = \mathrm{A}_{12x12} \cdot \mathrm{s(k-1)}_{12x1} + \omega(\mathrm{k-1})_{12x1}$$

$$\mathrm{z(k)}_{6x1} = \mathrm{H}_{6\mathrm{x}12} \cdot \mathrm{s(k)}_{12\mathrm{x}1} + \nu(\mathrm{k})_{6\mathrm{x}1}$$

Donde:

- A: es la matriz de transición.
- ω(k-1): variable aleatoria que representa el ruido del proceso (cambio de estado).
- H: es la matriz de observación.
- ν(k): variable aleatoria que representa el ruido de la medida.

Las variables ω(k-1) y ν(k) representan los ruidos y perturbaciones de los datos de detección de movimiento. Se asume que son independientes entre sí con distribuciones de probabilidad normales de media nula:

$$\mathrm{p}(\omega) \sim \mathrm{N}(0, \mathrm{Q})$$

$$\mathrm{p}(\nu) \sim \mathrm{N}(0, \mathrm{R})$$

Donde:

- Q: es la matriz de covarianza del ruido del proceso.
- R: es la matriz de covarianza del ruido de la medida.

Las matrices Q y R se definen como matrices diagonales cuyos elementos evolucionan de acuerdo con el filtro. Los valores con los que se inicializa son los siguientes:

$$Q_{12x12} = \begin{bmatrix} 0.001 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0.001 \end{bmatrix}$$

$$R_{6x6} = \begin{bmatrix} 0.999 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0.999 \end{bmatrix}$$

La variable de estado s(k) se define como un vector de 12 dimensiones que se muestra en la siguiente ecuación:

$$s(k) = \left[x(k), y(k), z(k), w(k), h(k), d(k), \sigma_{R1}(k), \sigma_{R2}(k), \sigma_{R3}(k), \sigma_{Q1}(k), \sigma_{Q2}(k), \sigma_{Q3}(k)\right]^T$$

Donde:

- x(k), y(k), z(k): son las coordenadas del centroide del objeto actual.
- w(k), h(k), d(k): representa las dimensiones de la *boundingbox* del objeto actual.
- $\sigma_{R1}(k)$, $\sigma_{R2}(k)$, $\sigma_{R3}(k)$: son los valores de covarianza de la matriz R.
- $\sigma_{Q1}(k)$, $\sigma_{Q2}(k)$, $\sigma_{Q3}(k)$: son los valores de covarianza de la matriz Q.

Como el intervalo de tiempo entre frames va a ser pequeño, el movimiento puede considerarse uniforme, y la matriz de transición A_{12x12} puede definirse constante:

	1	2	3	4	5	6	7	8	9	10	11	12
1	1	0	0	0	0	0	1	0	0	0	0	0
2	0	1	0	0	0	0	0	1	0	0	0	0
3	0	0	1	0	0	0	0	0	1	0	0	0
4	0	0	0	1	0	0	0	0	0	1	0	0
5	0	0	0	0	1	0	0	0	0	0	1	0
6	0	0	0	0	0	1	0	0	0	0	0	1
7	0	0	0	0	0	0	1	0	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0
9	0	0	0	0	0	0	0	0	1	0	0	0
10	0	0	0	0	0	0	0	0	0	1	0	0
11	0	0	0	0	0	0	0	0	0	0	1	0
12	0	0	0	0	0	0	0	0	0	0	0	1

La variable de medidas z(k) se emplea para observar tanto la posición del centroide de los objetos como las dimensiones del cubo que los delimita, y se define como un vector de 6 dimensiones que se muestra en la siguiente ecuación:

$$z(k) = [x(k), y(k), z(k), w(k), h(k), d(k)]^T$$

La matriz de observación H_{6x12} se define como:

	1	2	3	4	5	6	7	8	9	10	11	12
1	1	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0	0	0
5	0	0	0	0	1	0	0	0	0	0	0	0
6	0	0	0	0	0	1	0	0	0	0	0	0

4.2.1.1. Operación

El algoritmo del filtro de Kalman se basa en una etapa de predicción y otra de corrección recursivas que se relacionan según el siguiente diagrama:

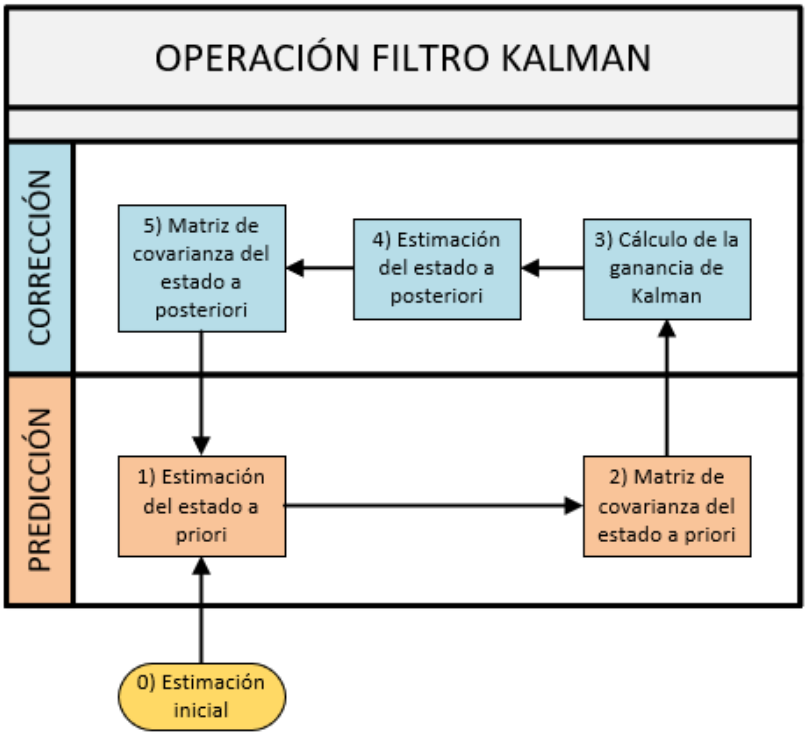

FIGURA 4.2-2: Esquema de operación del filtro de Kalman.

0) Estimación inicial: del estado $\hat{s}(0)$ y de la matriz de covarianza del estado $P(0)$

$$P(0)_{12x12} = \begin{bmatrix} 0.01 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0.01 \end{bmatrix}$$

1) Estimación del estado a priori: $\boxed{\hat{s}(\bar{k}) = A \cdot \hat{s}(k-1)}$

2) Matriz de covarianza del estado a priori: $\boxed{P(\bar{k}) = A \cdot P(k-1) \cdot A^T + Q}$

3) Cálculo de la ganancia de Kalman: $\boxed{K(k) = P(\bar{k}) \cdot H^T \cdot \left[H \cdot P(\bar{k}) \cdot H^T + R\right]^{-1}}$

4) Estimación del estado a posteriori: $\boxed{\hat{s}(k) = \hat{s}(\bar{k}) + K(k) \cdot \left[z(k) - H \cdot \hat{s}(\bar{k})\right]}$

5) Matriz de covarianza del estado a posteriori: $\boxed{P(k) = [I - K(k) \cdot H] \cdot P(\bar{k})}$

La ganancia de Kalman tiene dos objetivos: por un lado, pondera el tamaño del error de covarianza del estado a priori y de la matriz de covarianza del ruido de medida para determinar el mejor modelo entre la predicción y la observación; y, por otro lado, transforma la forma de representación de los residuos (diferencia entre medida predicha y la actual) del dominio de observación al dominio de estado.

4.2.1.2. Implementación

Para implementar el filtro de Kalman se hará uso de la clase *cv::KalmanFilter* de OpenCV. A cada uno de los objetos que aparezcan en la escena le corresponderá un filtro de Kalman para realizar su seguimiento. Para realizar una correcta asociación de datos, es decir, qué filtro le corresponde a cada objeto en cada instante de tiempo, se comparará la distancia euclídea de la predicción del estado actual con la observación actual del objeto, asociando aquel que se encuentre más cercano.

Además, para gestionar el problema de la oclusión se aplica un tiempo de vida a los filtros que va decrementando en cada paso de tiempo que no haya sido detectado. De este modo si un objeto se perdiera durante unos segundos tendría la posibilidad de recuperarse.

En el siguiente diagrama de flujo se muestra cómo se gestionan los filtros de Kalman de cada objeto:

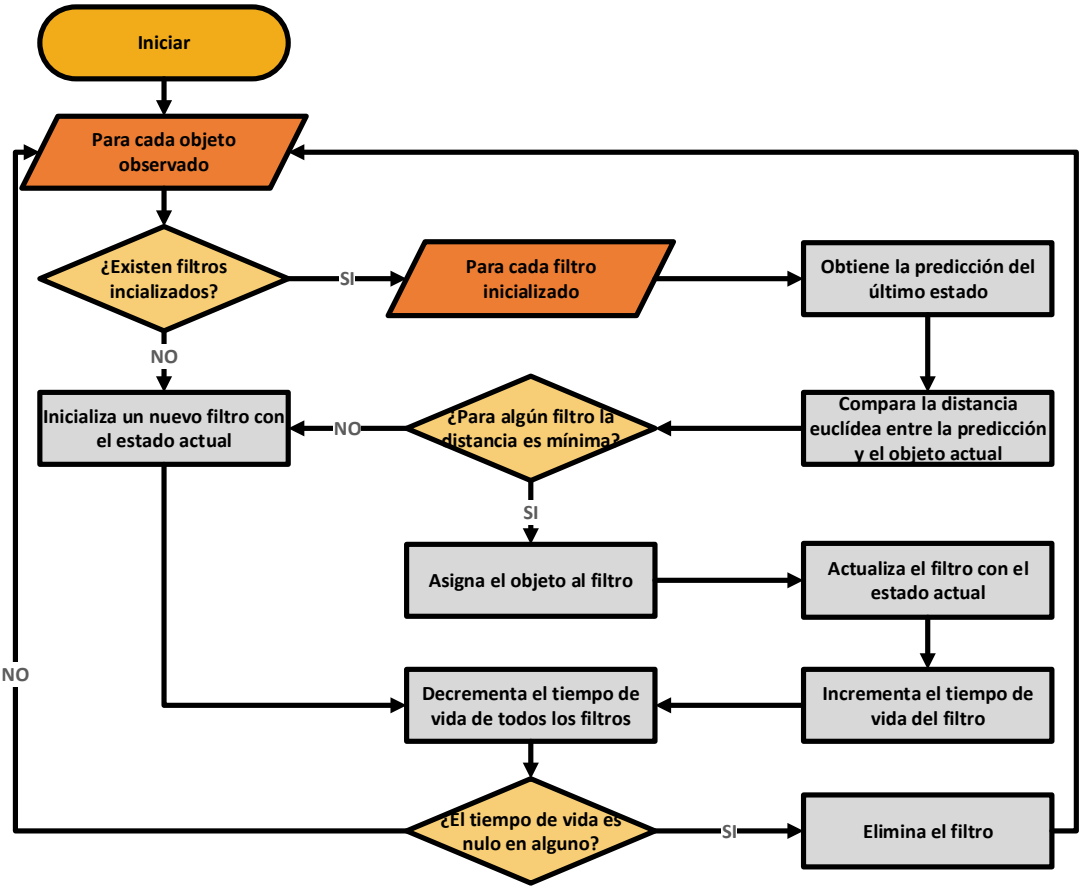

FIGURA 4.2-3: Esquema de implementación filtro de Kalman.

En las siguientes imágenes puede observarse los resultados obtenidos al aplicar el filtro de Kalman:

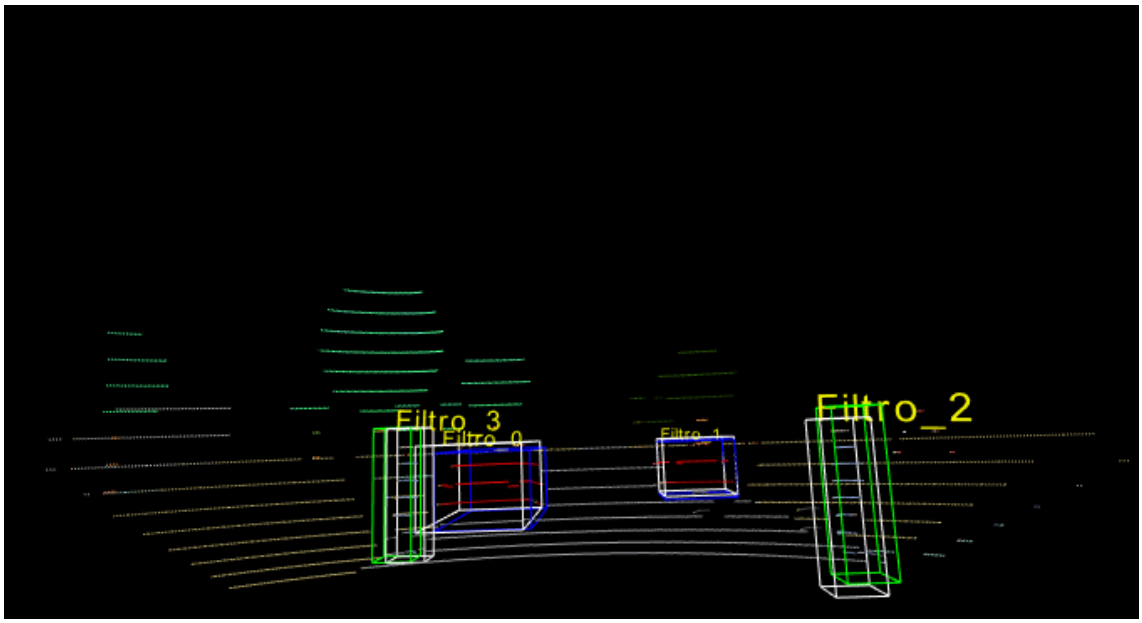

FIGURA 4.2-4: Ejemplo de implementación filtro de Kalman (punto de vista conductor)

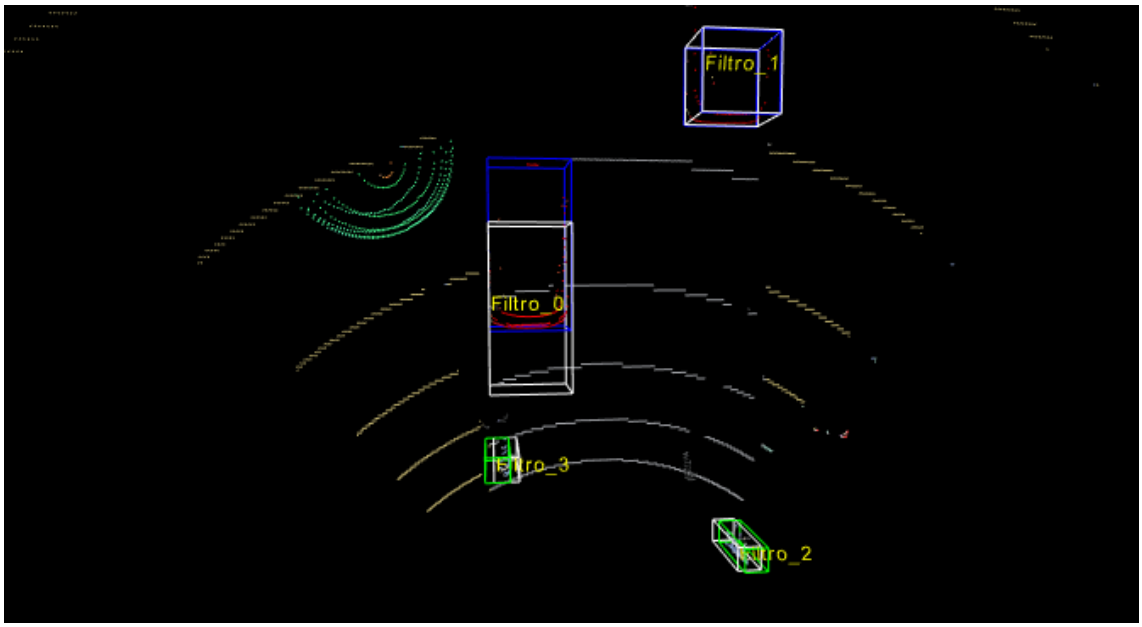

FIGURA 4.2-5: Ejemplo de implementación filtro de Kalman (vista cenital)

4.2.2.Precisión-tracking

La técnica de seguimiento mediante el paquete *precision-tracking* [1] emplea la forma 3D del objeto utilizando un marco probabilístico en el que hace uso de la información de la forma, el color y el movimiento, lo que aporta robustez cuando existen problemas de oclusión, cambios en el punto de vista u objetos lejanos con escasa información.

A diferencia del método de seguimiento que se ha utilizado para el filtro de Kalman, basado en un modelo de obstáculo con unas dimensiones de una *boundingbox* predefinidas, este método se basa en una rejilla para muestrear las velocidades del espacio de estados. A pesar de que los métodos basados en rejillas suelen emplearse en técnicas de SLAM (*Simultaneous Localization And Mapping*) y son lentos para el seguimiento de múltiples objetos, este enfoque permite hacer un muestreo fino en una rejilla grande gracias al uso de un método mediante histogramas denominado *Annealed Dynamic Histograms* (ADH).

ADH empieza muestreando el espacio de estados a una resolución gruesa calculando para cada muestra la probabilidad del estado en base a un modelo Bayesiano, y después subdivide algunas de las celdas de la rejilla para refinar la resolución. De este modo, con el paso del tiempo se va fortaleciendo (*annealed*) la distribución de probabilidad, acercándose a la verdadera.

4.2.2.1. Modelo Probabilístico

El modelo probabilístico empleado consiste en una Red Bayesiana Dinámica (RBD) que es un tipo de red bayesiana basada en un grafo acíclico dirigido (Directed Acyclic Graph, DAG)[5] para modelar estructuras de datos secuenciales o series temporales.

Las redes bayesianas implementan un modelo gráfico formado por un conjunto de variables (nodos) y sus relaciones de dependencia (arcos), que permite estimar la probabilidad posterior de variables no conocidas a partir de las que sí lo son.

La RBD consiste en una representación de los estados del proceso en un tiempo (red estática) y las relaciones temporales entre esos procesos (red de transición). Para una RBD suelen hacerse dos suposiciones:

- Proceso markoviano: el estado actual sólo depende del estado anterior.
- Proceso estacionario en el tiempo: las probabilidades condicionales en el modelo no cambian con el tiempo.

La RBD que se emplea para esta técnica de seguimiento es la siguiente:

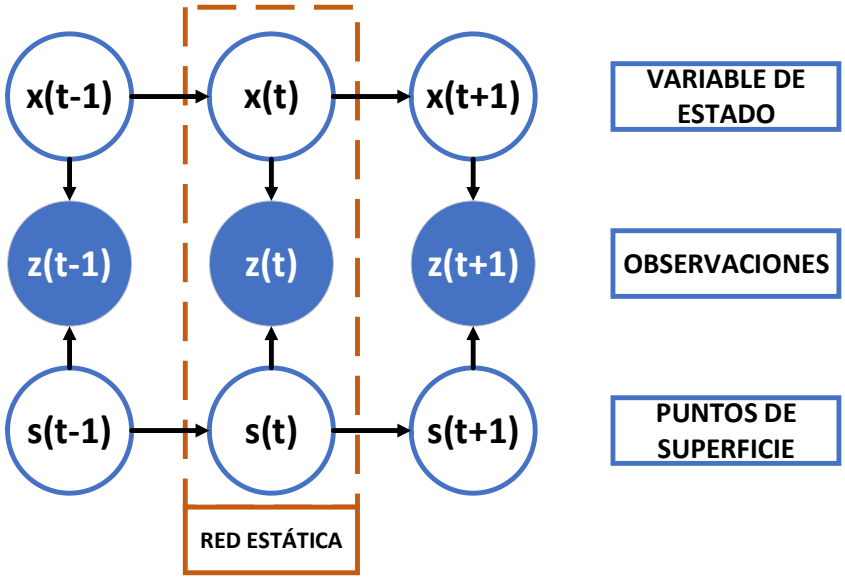

FIGURA 4.2-6: Red Bayesiana Dinámica del modelo empleado en precision-tracking.

[5] Se trata de un DAG porque no existe un camino de dependencias entre nodos que permita tener como origen y final el mismo nodo.

- Estado: la variable de estado que emplea el modelo probabilístico se define como:

$$x_t = (x_{t,p}, \dot{x}_t)$$

Donde:

- $x_{t,p}$: es la posición la posición lineal del objeto.
- $\dot{x}_t$: es la velocidad del objeto.

La posición mide el desplazamiento relativo desde el centroide del objeto en la última observación hasta la actual, y la velocidad incluirá sólo la componente lineal (la rotación es mínima en relación con la diferencia de tiempo entre frames). Además, se considera que los objetos se desplazan en el suelo, eliminando la componente vertical.

- Puntos de superficie: se emplean para combinar la forma 3D del objeto con el color y el movimiento y forman parte de la superficie visible del objeto.

- Observaciones (medidas): cada observación se genera a partir de la superficie y la variable de estado. Para cada punto de superficie se añade un ruido gaussiano que es función del ruido del sensor (Σe) que lo desplaza según el estado actual, generando la medida. Como las medidas anteriores se centran en el sistema de coordenadas, puede eliminarse la dependencia con el estado actual:

$$p(z_{t-1} | x_t, s_{t-1}) = p(z_{t-1} | s_{t-1})$$

Como pueden aparecer oclusiones o cambios del punto de vista, la probabilidad de obtener la superficie, conocida la del instante anterior se descompone en función de si el objeto era visible (V) o estaba ocluido ($\overline{V}$):

$$p(s_{t,j} | s_{t-1}) = p(V) \cdot p\left(s_{t,j} | s_{t-1}, V\right) + p(\overline{V}) \cdot p\left(s_{t,j} | s_{t-1}, \overline{V}\right)$$

Donde:

- $p\left(s_{t,j} | s_{t-1}, V\right)$: se modela como un gaussiano dependiente de la varianza de la resolución del sensor y la deformación del objeto, Σr, que varía con la distancia.

- $p\left(s_{t,j} | s_{t-1}, \overline{V}\right)$: se modela suponiendo que toda región no visible se encuentra ocluida, añadiendo unas constantes, k1, k2:

$$p\left(s_{t,j} | s_{t-1}, \overline{V}\right) = k_1 \cdot \left[k_2 - p\left(s_{t,j} | s_{t-1}, V\right)\right]$$

Por tanto, puede simplificarse la ecuación como:

$$p(s_{t,j} | s_{t-1}) = \eta \cdot \left[p\left(s_{t,j} | s_{t-1}, V\right) + k\right]$$

Donde:

- $\eta = p(V) - p(\overline{V}) \cdot k_1$
- $k = p(\overline{V}) \cdot k_1 \cdot k_2 / \eta$

4.2.2.2. Seguimiento

El objetivo del seguimiento es obtener el estado actual a partir de las observaciones pasadas. Usando la regla de Bayes puede escribirse como:

$$p(x_t|\, z_1 \dots z_t) = \eta \cdot p(z_t|x_t, z_1 \dots z_{t-1}) \cdot p(x_t|z_1 \dots z_{t-1})$$

Con:

- $p(x_t|z_1 \dots z_{t-1})$: modelo de movimiento que se ajusta a un gaussiano multivariante (modelo de velocidad constante mediante filtro de Kalman).

- $p(z_t|x_t, z_1 \dots z_{t-1})$: modelo de medida, del que no puede eliminarse la dependencia de las observaciones mediante Bayes al proceder del mismo objeto, pero que puede simplificarse incluyendo solo la dependencia de la última, a pesar de la pérdida de precisión:

$$p(z_t|x_t, z_1 \dots z_{t-1}) \approx p(z_t|x_t, z_{t-1})$$

  Aplicando la regla de la cadena, considerando las suposiciones de independencia e incluyendo los términos constantes en η, el modelo de medida puede escribirse como:

$$p(z_t|x_t, z_{t-1}) = \iint p(z_t, s_t, s_{t-1}|x_t, z_{t-1})\, ds_{t-1} ds_t =$$

$$= \eta \cdot \left[\prod_{z_j \,\epsilon\, z_t} \exp\left(-\frac{1}{2} \cdot \left(z_j - \overline{z}_i\right)^T \cdot \Sigma^{-1} \cdot \left(z_j - \overline{z}_i\right) \right) + k \right]$$

  Donde:

  - $\Sigma = 2\Sigma_e + \Sigma_r$
  - $\overline{z}_i \,\epsilon\, \overline{z}_{t-1} = z_{t-1} + x_{t.p}$ (es el punto más cercano).

4.2.2.3. Modelo de color

Es posible incluir al modelo la información de color tomando la distribución de probabilidad del color con el objetivo de alinear los puntos correctamente, y así construir un conjunto de correspondencias entre frames consecutivos. Se construye un histograma normalizado con las diferencias de valores de color entre puntos, que sigue una distribución laplaciana (la probabilidad de cambio de color entre frames será menor cuanto mayor sea el valor del color).

Para simplificar el proceso de aprendizaje, se simplifica el modelo de color considerando únicamente el canal de color azul que es con el que obtienen un mejor rendimiento.

El color se incorpora al modelo de medida, como un producto de gaussianas:

$$p(s_{t,j}|\, s_{t-1}, V) = p_s(s_{t,j}|\, s_{t-1}, V) \cdot p_c(s_{t,j}|\, s_{t-1}, V) =$$

$$= p_s(s_{t,j}|\, s_{t-1}, V) \cdot \left[p(C) \cdot p(s_{t,j}|\, s_{t-1}, V, C) + p(\overline{C}) \cdot p(s_{t,j}|\, s_{t-1}, V, \overline{C})\right]$$

Con:

- $p_s(s_{t,j}|\, s_{t-1}, V)$: probabilidad de coincidencia basada en la posición.
- $p_c(s_{t,j}|\, s_{t-1}, V)$: probabilidad de coincidencia basada en el color.
- $p(C)$ y $p(\overline{C})$: probabilidades de que el color del frame actual coincida o no, respectivamente, con el modelo de color aprendido.
- $p(s_{t,j}|\, s_{t-1}, V, C)$: distribución de color (parametrizado como un laplaciano).
- $p(s_{t,j}|\, s_{t-1}, V, \overline{C}) = 1/255$: probabilidad de que un punto tenga cualquier color.

El parámetro p(C) depende de la resolución del muestreo, ya que será más preciso cuanto más se refine, por esto, debe ser una función de este.

$$p(C) = p_c \cdot \exp\left(-\frac{r^2}{2\sigma_c^2}\right)$$

Donde:

- r: es la resolución de muestreo
- σc: es el parámetro que controla la velocidad de aumento de la probabilidad con el decremento de la resolución hasta r=0.

Añadiendo el modelo de color al modelo de medida completo se obtiene finalmente:

$$p(z_t|x_t, z_{t-1}) = \eta \cdot \left[\prod_{z_j \,\epsilon\, z_t} p_s(z_j|x_t, z_{t-1}) \cdot p_c(z_j|x_t, z_{t-1}, V) + k_3 \cdot [k_4 - p_s(z_j|x_t, z_{t-1})]\right]$$

Donde:

- k_3=k/(k+1)
- k_4=1 (parámetro elegido por validación cruzada)

4.2.2.4. Annealed Dynamic Histograms (ADH)

Calcular cada uno de los modelos anteriores para cada estado considerado hace que el método sea demasiado lento. Por eso se propone la técnica ADH, que consiste en dividir el espacio de estados en una rejilla de forma gruesa y calcular la probabilidad del estado para cada celda para luego subdividir algunas de las celdas de manera recursiva.

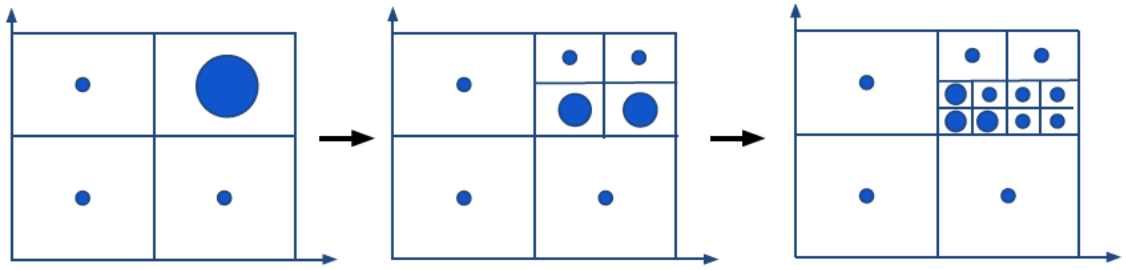

FIGURA 4.2-7: Descomposición del espacio de estados usando ADH [1].

La probabilidad de cada una de las celdas viene dada por la siguiente ecuación:

$$p(c_i) = p(c_i \cap R) = \eta \cdot p(x_i|z_1 \dots z_t) \cdot p(R)$$

Donde:

- xi: es el estado en cada una de las celdas ci.
- R: es el conjunto de todas las celdas que se dividen en ci.
- η: constante que sólo depende de las celdas en R.

El criterio que se emplea para decidir qué celdas dividir se basa en minimizar la diferencia entre la función de distribución antes y después de realizar la división. Para comparar las distribuciones se emplea la Divergencia de Kullback-Leibler (D_{KL}) que es una medida no simétrica de la similitud o diferencia entre dos funciones de probabilidad en la que una representa la distribución verdadera y la otra una aproximación.

Tras la división las nuevas subceldas pueden adoptar probabilidades separadas que permiten una aproximación más precisa a la verdadera. La máxima divergencia se dará cuando una subcelda tenga la misma probabilidad que la celda completa, siendo el resto nulas, y viene dada por:

$$\max[D_{KL}(A||B)] = \max\left[\sum_{j=1}^{k} p_j \cdot \ln\left(\frac{p_j}{P_i/k}\right)\right] = P_i \cdot \ln(k)$$

Donde:

- B: es la distribución antes de dividir la celda.
- A: es la nueva distribución obtenida de dividir una celda en k subceldas.
- Pi: es la probabilidad de la celda que se va a dividir.
- pj: es la probabilidad de cada una de las subceldas una vez hecha la división.

Para encontrar el histograma más coincidente basta con dividir todas las celdas que superen cierto umbral de probabilidad (pmin), y para obtener la máxima eficiencia por unidad de tiempo debe elegirse un k pequeño. En este caso han decidido que cada dimensión se divida en tres partes (k=3).

En las primeras iteraciones, al ser muy gruesa la resolución, la alineación entre un frame y el anterior no será correcta, y se introducirá una nueva fuente de error, Σg, que aumenta la varianza del modelo de medida y será proporcional a la resolución de muestreo, reduciéndose a medida que se refina:

$$\Sigma = 2\Sigma_e + \Sigma_r + \Sigma_g$$

En el siguiente flujograma podemos ver el método ADH:

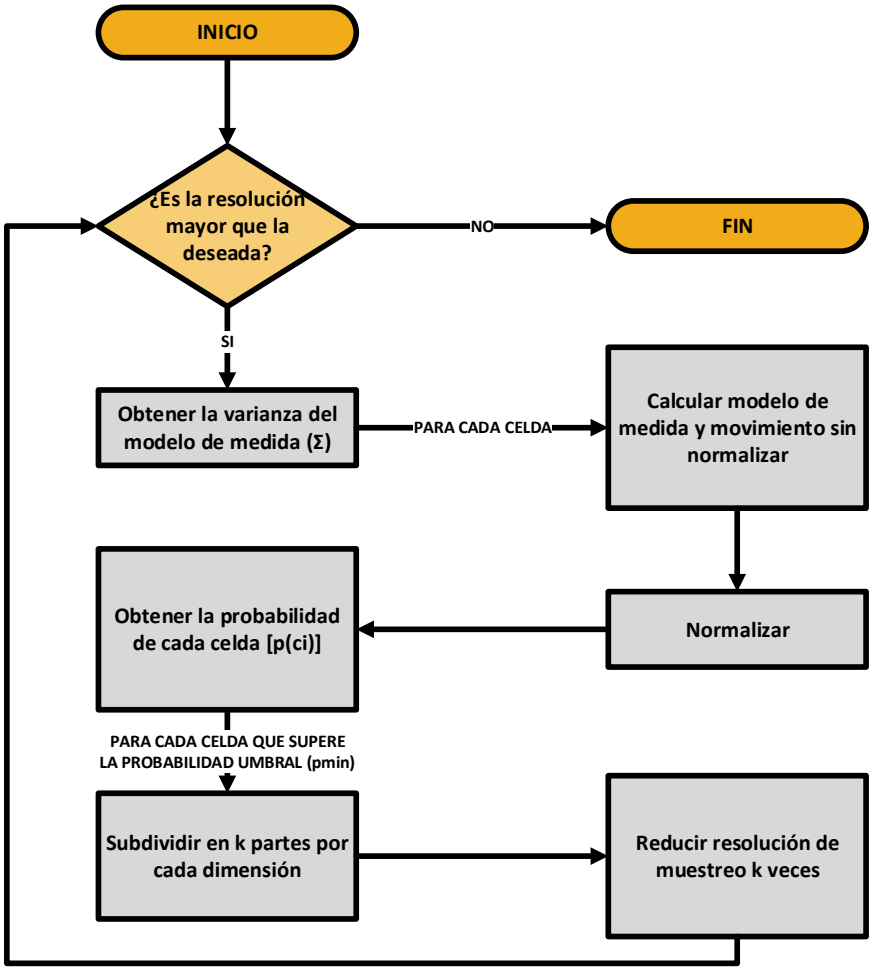

FIGURA 4.2-8: Método ADH.

4.2.2.5. Implementación

Previamente a su aplicación, se necesita utilizar el algoritmo de segmentación y asociación de datos basado en nubes de puntos, que segmenta los objetos en clústeres, el cual se ha presentado en apartados anteriores. El método de seguimiento precision-tracking estima la velocidad de cada uno de los objetos previamente segmentados.

Para su implementación primero deben definirse los parámetros que se han comentado en la explicación de los apartados anteriores. A continuación, se definen los más característicos y que han sido modificados[6]:

Parámetro	Descripción	Valor
kDesiredSamplingResolution	Resolución de muestreo deseada	0.05
kReductionFactor	Reducción de la resolución de muestreo en cada iteración	3
kMinProb	Probabilidad mínima para dividir una celda (pmin)	0.0001
kMinMeasurementVariance	Covarianza del ruido del sensor independiente de la distancia al objeto. (2Σe)	0.03m
kSmoothingFactor	Factor de suavizado que evita una probabilidad nula de los puntos que no se alinean. (k)	0.8
kMaxDiscountPoints	Máximo número de puntos independientes por objeto, si se supera, se descarta el modelo de medida.	150
kProbColorMatch	Probabilidad de color (pc)	0.05
kColorSpace	Espacio de color que se usa para la coincidencia de color. (0-BG, 1-R+G+B/3)	1
useColor	Determina si se emplea el color (ralentiza)	True
use3D	Determina si se sigue la nube completa 3D o una proyección 2D	True
kCurrFrameDownsample	Número de puntos del frame actual del objeto	300
kPrevFrameDownsample	Número de puntos del frame anterior del objeto	500
stochastic_downsample	Determina si se realiza un muestro determinístico o estocástico del objeto	True
kInitialXYSamplingResolution	Resolución inicial de muestreo	1m

TABLA 4.2-1: Parámetros del algoritmo precision-tracking.

[6] Para más información acudir a [1].

Cada vez que aparece un nuevo objeto que seguir, es decir, que el filtro de Kalman no se ha podido asociar con ninguno existente, se crea un nuevo tracker.

```
precision_tracking::Tracker tracker(&params);
tracker.setPrecisionTracker(boost::make_shared<precision_tracking::PrecisionTracker>(&para
ms));
```

Por cada objeto visualizado, se obtiene la resolución del láser en función de la distancia, y se añade la nube de puntos del objeto para obtener su velocidad estimada.

```
precision_tracking::getSensorResolution(objetos[n_obj].centroid,
&sensor_horizontal_resolution, &sensor_vertical_resolution);
tracker.addPoints(pointcloud, timestamp, sensor_horizontal_resolution,
sensor_vertical_resolution, &estimated_velocity);
```

El cálculo de la resolución horizontal y vertical se realiza aplicando la siguiente ecuación:

$$\mathrm{Hres[m]} = 2 \cdot \mathrm{d_H} \cdot \tan\left(\frac{\mathrm{Hres[º]}}{2} \cdot \frac{\pi}{180}\right)$$

$$\mathrm{Vres[m]} = 2 \cdot \mathrm{d_H} \cdot \tan\left(\frac{\mathrm{Vres[º]}}{2} \cdot \frac{\pi}{180}\right)$$

Donde:

- $\mathrm{d_H} = \sqrt{x^2 + y^2}$: es la distancia al objeto seguido
- $\mathrm{Hres[º]} = 0.2$: es la resolución horizontal según especificaciones técnicas.
- $\mathrm{Vres[º]} = 2$: es la resolución vertical según especificaciones técnicas.

Capítulo V

SISTEMA REAL

En este apartado se expondrán los puntos de la arquitectura del sistema real que difieren respecto a la estructura de simulación explicada en apartados anteriores.

5.1. Entorno de adquisición de datos

La diferencia más significativa respecto a la simulación es el origen de la información. Mientras que en simulación obteníamos directamente los datos de los sensores simulados, en el caso real, la información tanto de LiDAR, como de cámara y GPS se han adquirido y han sido guardadas en un archivo de ROS *bag*. La escena de grabación es la misma implementada en VREP, de este modo podremos realizar una comparativa visual de los resultados obtenidos.

FIGURA 5.1-1: Imagen de la escena adquirida por la cámara.

5.2. Marcos de coordenadas

En el caso real, la ubicación de los sensores no será la misma que para el caso de simulación. Podemos visualizar el árbol de transformadas utilizando la herramienta rqt, que mostrará el siguiente esquema:

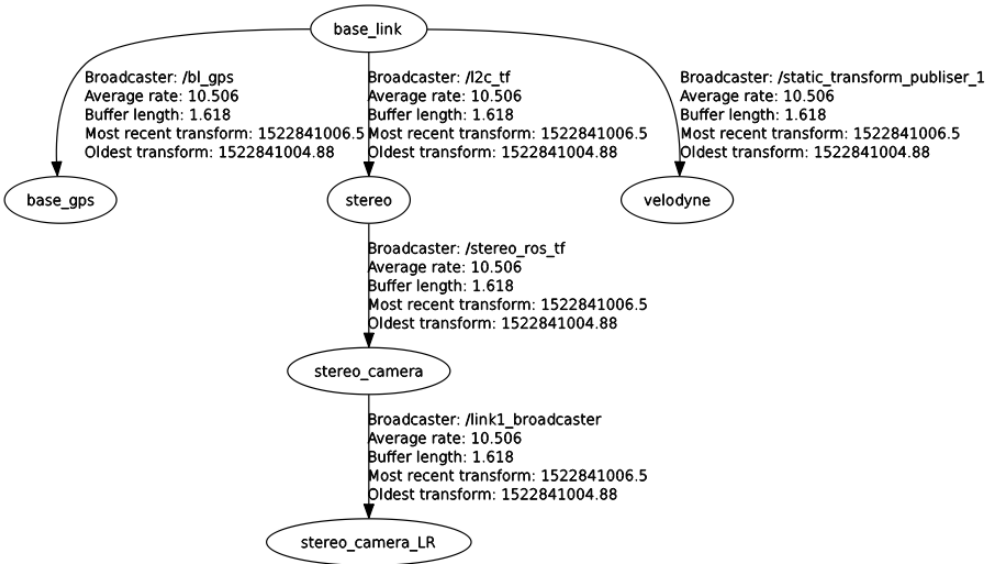

FIGURA 5.2-1: Árbol de transformadas del sistema real.

En el grafo se puede ver que el marco de referencia es *base_link*, al que se encuentran referenciadas las coordenadas de los sensores (*base_gps*, *stereo* y *velodyne*). Todas las relaciones entre los diferentes marcos de coordenadas son generadas a través del launch *genera_tf_real.launch*.

En la siguiente tabla e imágenes podemos apreciar cuál es la relación entre los marcos en valores numéricos de traslación y rotación, y su ubicación sobre el vehículo, que podemos visualizar en rviz:

Marco de coordenadas	**Posición Absoluta**			**Orientación Absoluta**			
	X	**Y**	**Z**	**R**	**P**	**Y**	**w**
base_link	0	0	0	0	0	0	1
velodyne	0	0	1	0	0	0	1
base_gps	-0.35	0	0.75	0	0	0	1
stereo	0.956	0.192	0.706	-0.022	0.173	0.051	0.983
stereo_camera	0.956	0.192	0.706	0.614	-0.542	0.39	-0.42
stereo_camera_LR	0.956	0.192	0.706	0.614	-0.542	0.39	-0.42

TABLA 5.2-1: Posición y orientación de los marcos de coordenadas en sistema real.

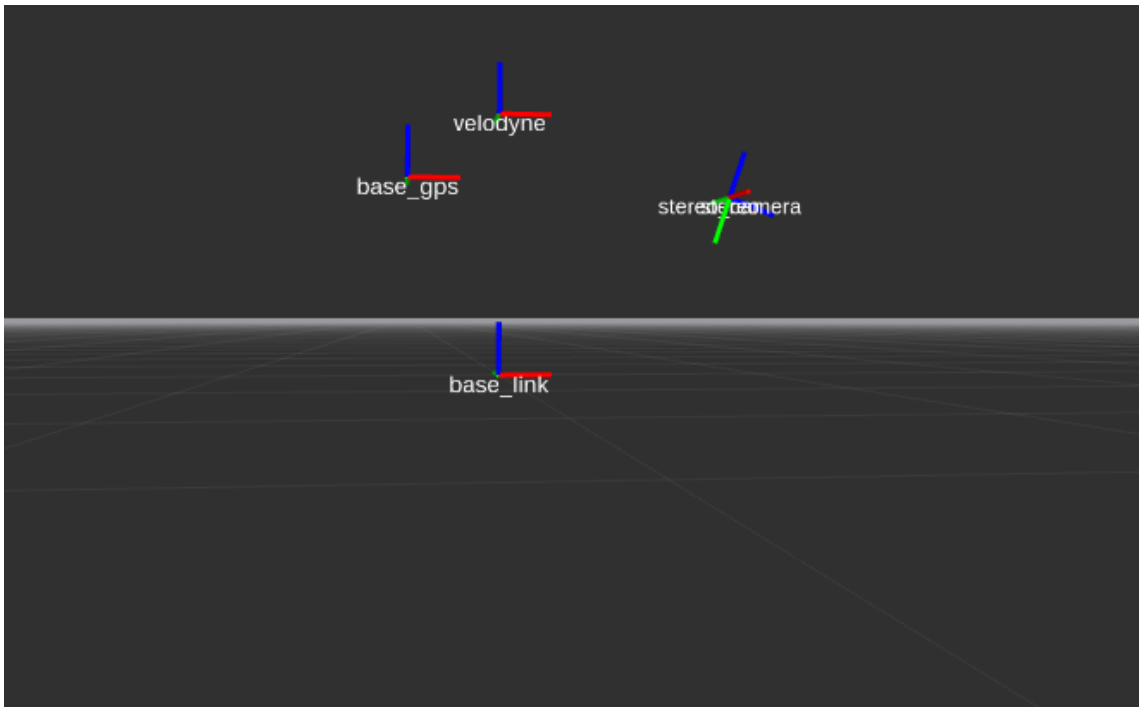

FIGURA 5.2-2: Marcos de coordenadas de sistema real en rviz (sin modelo).

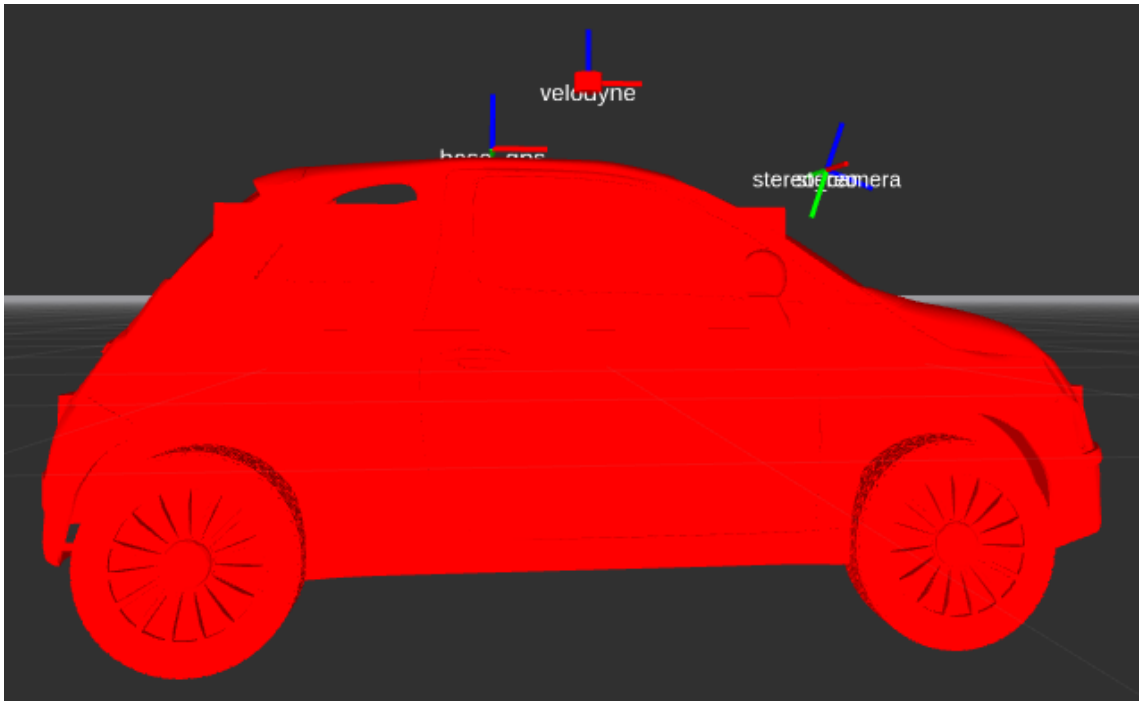

FIGURA 5.2-3: Marcos de coordenadas de sistema real en rviz (con modelo).

5.3. Calibración y fusión sensorial

Para realizar la etapa de fusión sensorial, a diferencia de simulación no se empleará el paquete but_velodyne, sino que se hará uso de un paquete similar denominado velo2cam_calibration [22] que implementa un algoritmo de calibración automática para sistemas con LiDAR y cámara estéreo.

El método empleado en velo2cam no depende de una configuración fija del sistema, por lo que las posiciones realtivas entre los sensores pueden ser muy diversas. Ademas, permite usar láseres de menos resolución (como en este caso de 16 haces).

El funcionamiento de velo2cam es similar al de but_velodyne, toma las imágenes procedentes de la cámara y del velodyne en sus respectivos marcos de coordenadas (*/stereo_camera_LR* y */velodyne*) y publica una nube de puntos coloreada.

Los resultados obtenidos de su implementación pueden observarse en las siguientes imágenes:

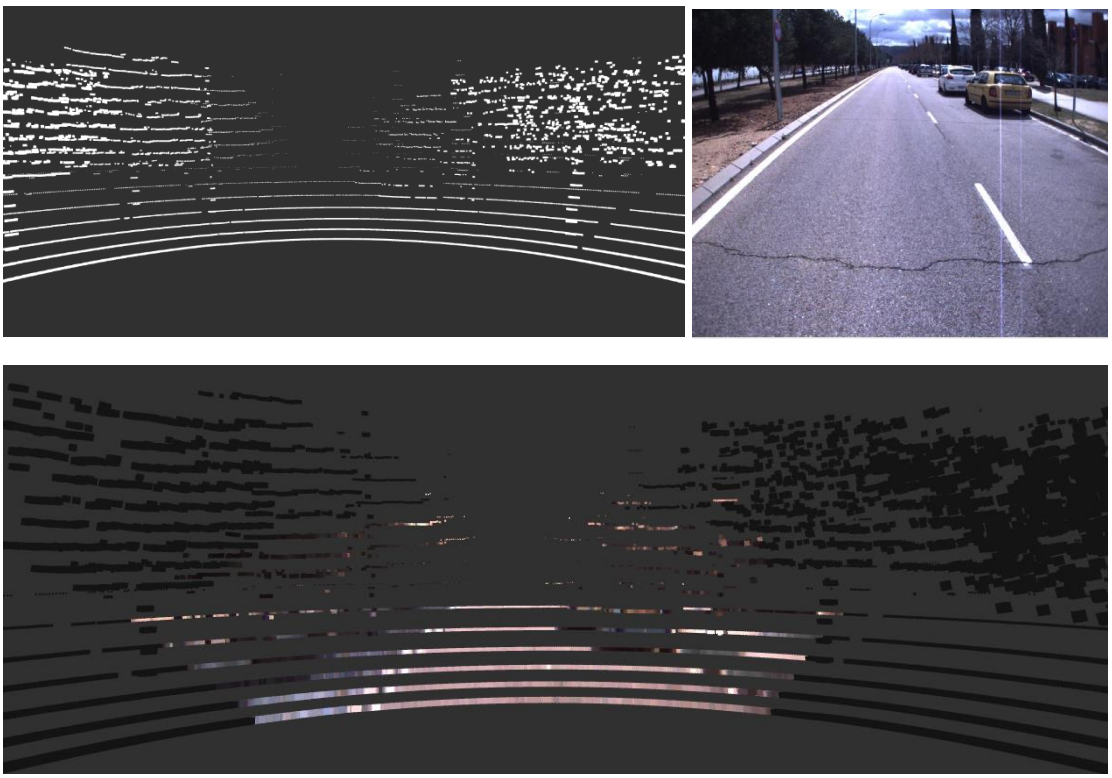

FIGURA 5.3-1: Proceso de fusión sensorial real.

5.4. Segmentación de objetos

Trabajar directamente con la imagen real obtenida de la cámara fusionada con la nube de puntos no aporta una gran información de color que nos permita distinguir cada uno de los objetos, como se puede apreciar en la imagen del apartado anterior. En simulación este hecho no resultaba un problema, ya que se realizaba una segmentación manual de los objetos aplicando el mismo color a todos los que pertenecían a la misma clase.

Para poder obtener un resultado similar en el caso real se ha empleado el resultado obtenido de una red neuronal entrenada[7] que permite realizar una segmentación semántica de los objetos de la escena diferenciando entre distintas clases:

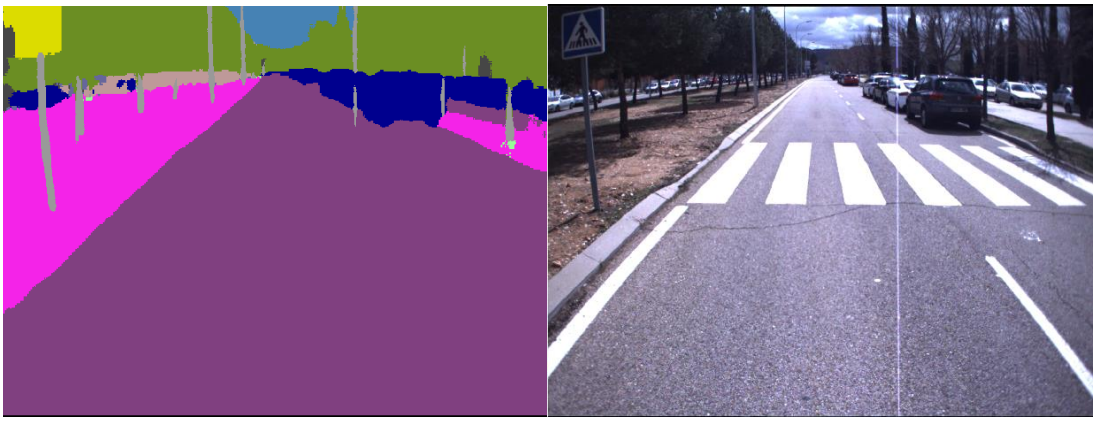

FIGURA 5.4-1: Comparativa de la imagen real frente a la segmentación semántica realizada con la red neuronal.

De la imagen segmentada podemos realizar la siguiente clasificación de colores:

Clase de objeto	Color	R	G	B
Suelo	Rosa	243	35	232
Carretera	Morado	128	63	127
Señales de tráfico	Amarillo	220	220	0
Cielo	Azul claro	70	130	180
Vehículos	Azul oscuro	0	0	140

TABLA 5.4-1: Códigos de colores para cada uno de los objetos segmentados.

En este caso la detección aplicará un filtro de color que empleará un rango de valores correspondientes al vehículo.

[7] La obtención de la imagen semántica no es objeto de este trabajo por lo que para más información se aconseja acudir a [23].

5.5. Obtención de la posición del GPS

Para poder obtener la posición y velocidad del coche en movimiento se obtendrán los datos del gps, que publica mensajes del tipo *sensor_msgs::NavSatFix*, que están en coordenadas globales de navegación (latitud, longitud, altitud), y se convertirán a coordenadas UTM[8] (geodesy::UTMPoint) que trabaja en coordenadas lineales métricas.

Para realizar esta conversión es necesario conocer la zona UTM en la que se ubica el GPS que viene determinada por un número y una letra como puede apreciarse en la siguiente imagen:

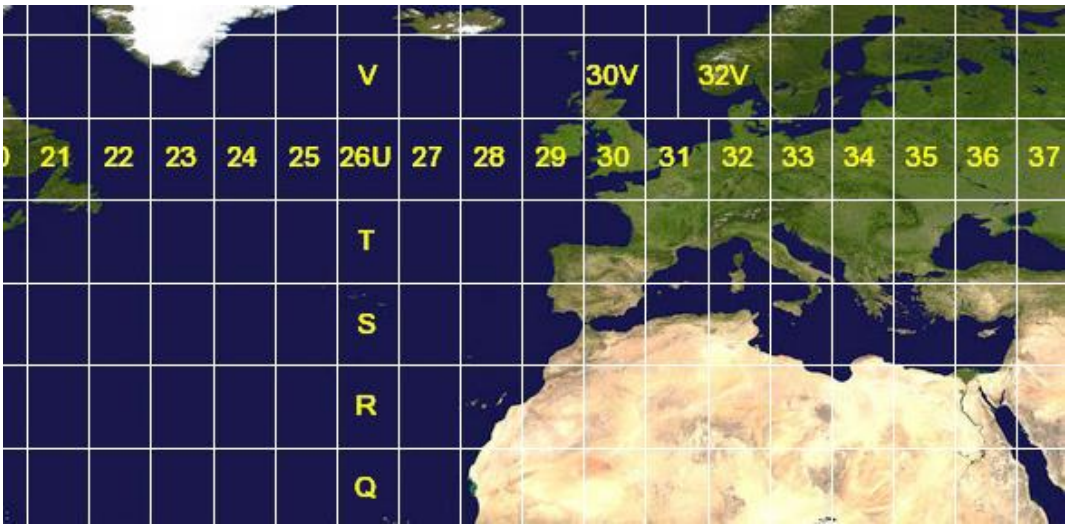

FIGURA 5.5-1: Mapa con coordenadas UTM.

Como puede apreciarse la ubicación del GPS es la zona 30T (Alcalá de Henares).

La conversión a coordenadas UTM se realiza en el nodo *odometry_and_tf.cpp* que publica la posición en el topic */odom* que deberá sincronizarse con la publicación de la nube de puntos coloreada para poder determinar correctamente la velocidad de desplazamiento en cada frame.

Partiendo de que se conoce la localización geográfica de origen, es posible obtener la posición del GPS por diferencia de coordenadas, como podemos ver en el siguiente extracto de código:

[8] Las coordenadas UTM son una proyección cartográfica del globo terráqueo en forma de cilindros que generan zonas UTM en forma de cuadrícula de longitud 6º.

```
//initialize origin point
double lat_origin = 40.5126566;
double lon_origin = -3.34460735;

//and convert it to UTM
geodesy::UTMPoint utm_origin(lat_origin, lon_origin, 30, 'T');

double lat_actual = gps_data.latitude;
double lon_actual = gps_data.longitude;

//and convert it to UTM
geodesy::UTMPoint utm_actual(lat_actual, lon_actual, 30, 'T');

///PUBLICAR DATOS DEL ODOM2

odom2.header.frame_id = "/base_link";
odom2.header.stamp = ros::Time::now();

odom.pose.pose.position.x=utm_actual.easting-utm_origin.easting;
odom.pose.pose.position.y=utm_actual.northing-utm_origin.northing;

odom_pub.publish(odom2);
```

Capítulo VI

EVALUACIÓN DE RESULTADOS

En este apartado se van a presentar los resultados obtenidos de la aplicación del método precision-tracking para el seguimiento de objetos y su comparativa con el seguimiento mediante filtro de Kalman, así como el resultado del sistema de detección de obstáculos implementado tanto en simulación como en un caso real.

6.1. Métricas de evaluación

La evaluación del sistema implementado en los casos propuestos se va a realizar de manera cualitativa mediante una interpretación visual del rendimiento en la detección de objetos; y de manera cuantitativa, mediante una comparación numérica de los resultados obtenidos comparados con el *groundtruth* del vehículo.

Para evitar contaminar la evaluación de los resultados con errores en la detección (falsos negativos), se van a descartar las siguientes medidas:

- Detecciones en las que la distancia euclídea entre centroides de la predicción y la detección supera los 2.5m.
- Diferencia de velocidad superior 5km/h con respecto al valor real.
- Diferencias de estimación de velocidad entre instantes consecutivos superiores al 50%.

Con esto, podemos obtener el número de falsos negativos y compararlos con los verdaderos positivos, mediante la tasa de falsos negativos porcentual (TFNP):

$$\boxed{\mathrm{TFNP} = \frac{\mathrm{FN}}{\mathrm{TP} + \mathrm{FN}} \cdot 100}$$

Donde:

- FN: son los falsos negativos eliminados siguiendo los criterios expuestos.
- TP: son los verdaderos positivos (1-FN)

Las métricas que se van a emplear para realizar la comparación entre las velocidades obtenidas son las que se exponen a continuación:

- Error Relativo Medio Porcentual (ERMP): es la media de la suma de los errores absolutos porcentuales entre la velocidad real y la estimada:

$$\mathrm{ERMP} = \frac{1}{\mathrm{N}} \cdot \sum \left(\frac{|\mathrm{V_{GT}(i)} - \mathrm{V_{SIS}(i)}|}{\mathrm{V_{GT}(i)}} \cdot 100 \right)$$

- Raíz Cuadrada del Error Cuadrático Medio (RECM): es la raíz cuadrada de la media de la suma de los errores cuadráticos entre la velocidad real y la estimada:

$$\mathrm{RECM} = \sqrt{\frac{1}{\mathrm{N}} \cdot \sum ([\mathrm{V_{GT}(i)} - \mathrm{V_{SIS}(i)}]^2)}$$

Donde:

- V_{GT}: es la velocidad real
- V_{SIS}: es la velocidad obtenida por el sistema de seguimiento
- N: es el número de iteraciones durante las que sigue al objeto.

6.2. Evaluación de la detección

Para evaluar si la detección realizada de los objetos es correcta, se verificarán cualitativamente los resultados obtenidos para diferentes situaciones. Se implementará de forma que por cada objeto detectado se cree un marcador visual esférico (*visualization_msgs::Marker*) en el centro del objeto que se publicará a través del topic */obstacle_pub_*, como puede observarse en el siguiente extracto de código:

```
visualization_msgs::Marker points;
geometry_msgs::Point p;
points.header.frame_id = "/base_link";
points.header.stamp = ros::Time::now();
points.ns = "Marcador";
points.action = visualization_msgs::Marker::ADD;
points.pose.orientation.w = 1.0;

points.id = 0;
points.type = visualization_msgs::Marker::SPHERE_LIST; //POINTS

// POINTS markers use x and y scale for width/height respectively
points.scale.x = 1; points.scale.y = 1; points.scale.z = 1;

// Points are yellow
points.color.r = 1.0f; points.color.g = 1.0f;  points.color.a = 1.0f;

points.points.clear();

for (int n_obj=0; n_obj<objetos.size(); n_obj++)
{
        p.x=objetos[n_obj].centroid(0);
        p.y=objetos[n_obj].centroid(1);
        p.z=0;
        points.points.push_back(p);
}

obstacle_pub_.publish(points);
```

A continuación, se mostrarán varios casos distintos tanto en simulación como en un caso real, donde podrá compararse la posición real de los objetos con el marcador de objeto publicado en rviz.

6.2.1. Casos simulación

6.2.1.1. Caso 0: Objeto individual

En primer lugar, se visualizará la detección para un solo objeto, en este caso un vehículo. A continuación, pueden observarse los resultados obtenidos:

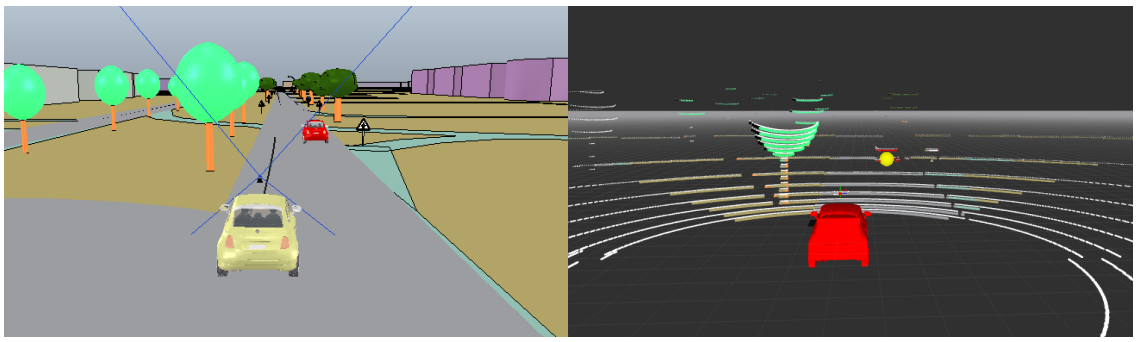

FIGURA 6.2-1: Detección para el caso de un objeto – caso 0.

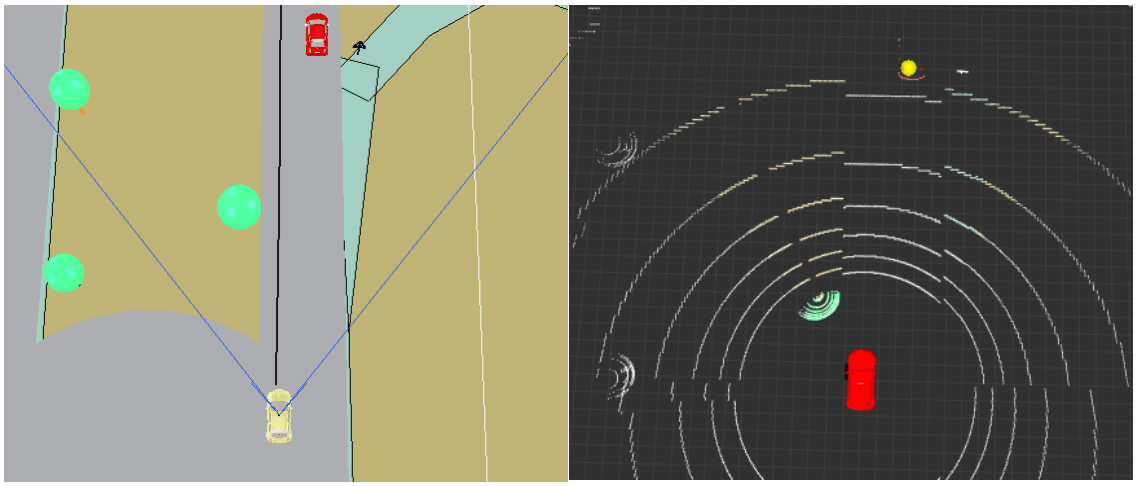

FIGURA 6.2-2: Detección para el caso de un objeto – caso 0 (vista cenital).

Podemos ver que el objeto es detectado correctamente.

6.2.1.2. Caso 1: Objetos múltiples

En segundo lugar, se visualizará la detección para varios objetos (tanto peatones como vehículos) en la misma escena anterior. A continuación, pueden observarse los resultados obtenidos, donde se aprecia que se detectan correctamente:

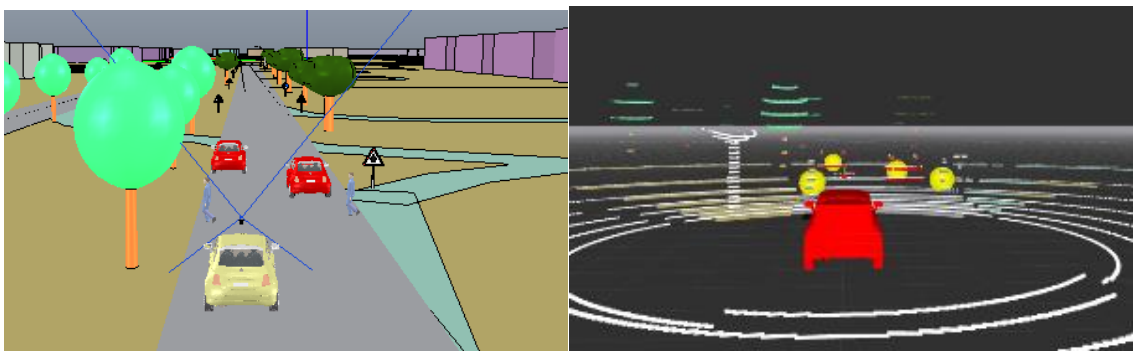

FIGURA 6.2-3: Detección para el caso de múltiples objetos – caso 1.

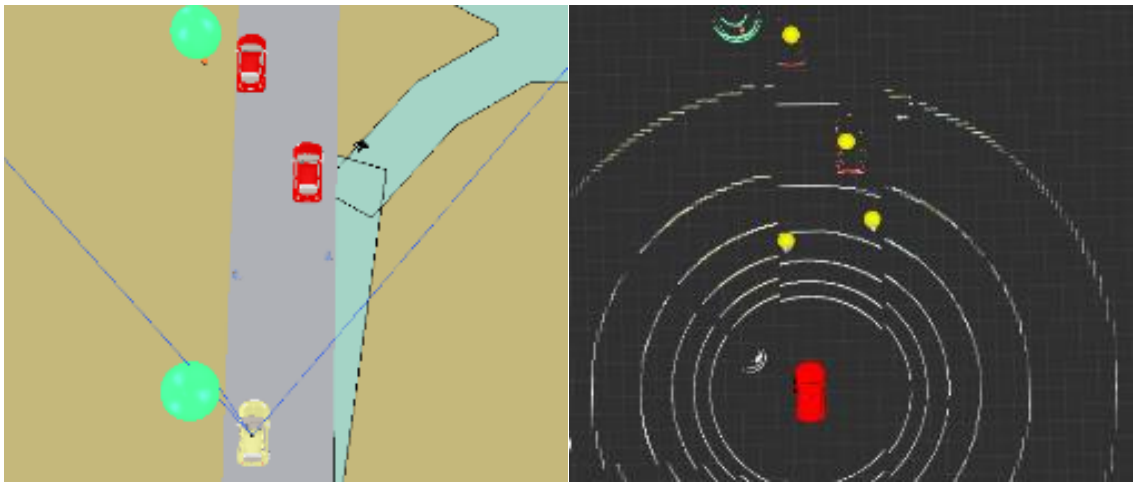

FIGURA 6.2-4: Detección para el caso de múltiples objetos – caso 1 (vista cenital).

6.2.1.3. Caso 2: Objetos múltiples orientaciones

Por último, se visualizará la detección para más objetos (tanto peatones como vehículos) en una escena distinta con diferentes orientaciones. A continuación, pueden observarse los resultados obtenidos:

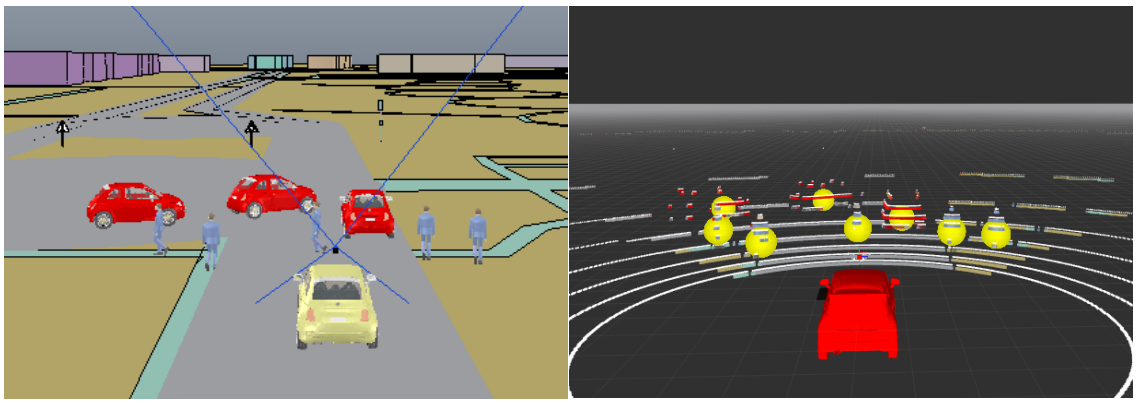

FIGURA 6.2-5: Detección para el caso de múltiples objetos – caso 2.

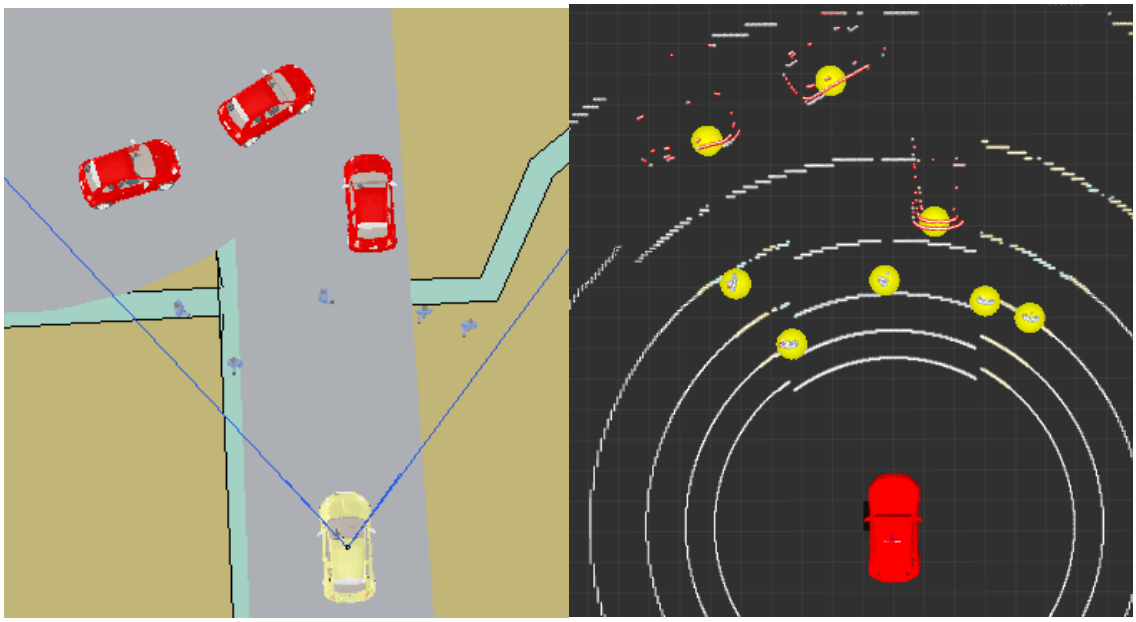

FIGURA 6.2-6: Detección para el caso de múltiples objetos – caso 2 (vista cenital).

Igual que en los casos anteriores, los objetos que fueron representados en la escena son detectados correctamente.

6.2.2. Caso real

En este caso se empleará un bag grabado para comparar los resultados obtenidos en una escena en la que el vehículo va desplazándose mientras se van detectando otros coches a los lados aparcados.

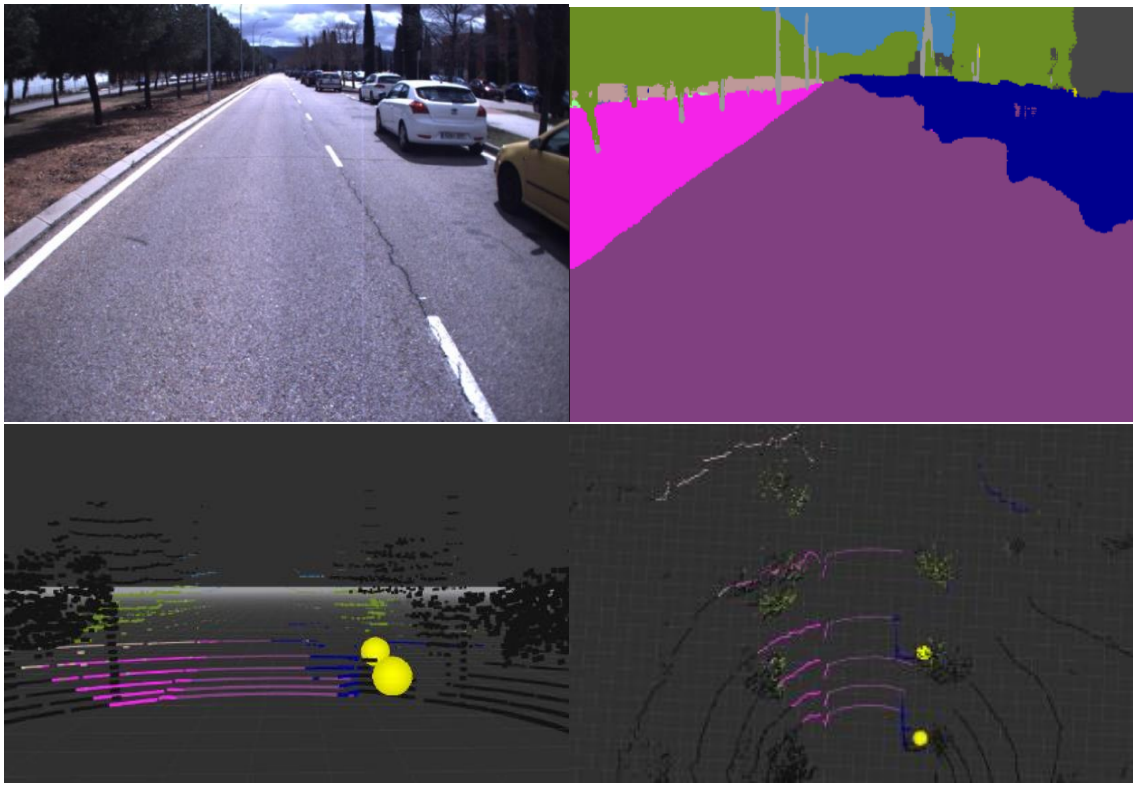

FIGURA 6.2-7: Detección para el caso real desde diferentes puntos de vista.

En la imagen podemos ver cómo se han detectado los dos vehículos más cercanos, pero no los más alejados.

6.3. Evaluación del seguimiento

6.3.1. Casos simulación

6.3.1.1. Caso 0: Objeto individual estático

En primer lugar, para poder medir la precisión en la estimación de la velocidad se realizará el seguimiento de un único vehículo estacionado, que tomando como marco de coordenadas el vehículo propio, tendrá una velocidad relativa cuyo valor absoluto es igual a la velocidad de desplazamiento propia.

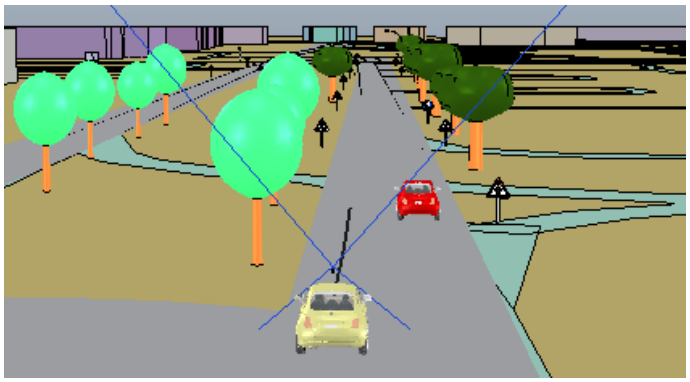

FIGURA 6.3-1: Escena de seguimiento caso objeto estático.

En las siguientes gráficas se muestra el error de velocidad obtenido en una de las pruebas:

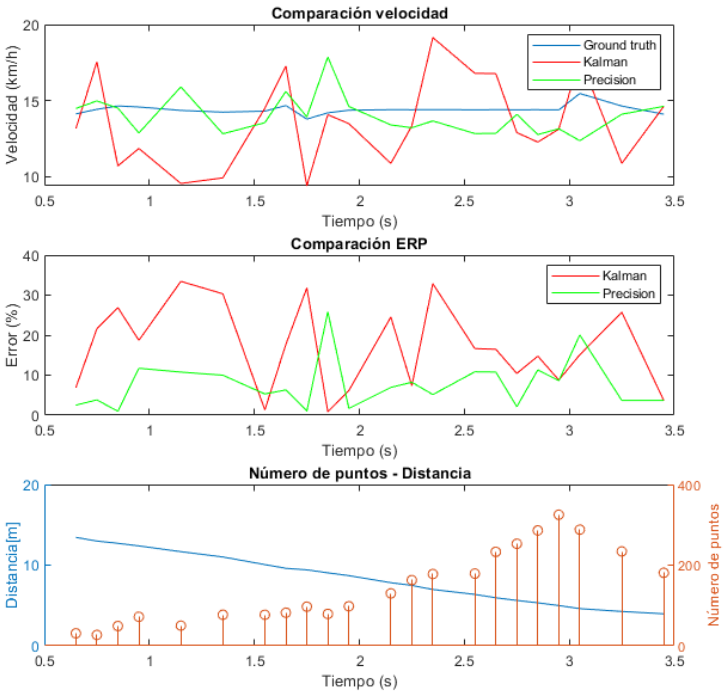

Figura 6.3-2 Gráfica de comparación de errores en el seguimiento de la velocidad caso objeto individual estático.

Como podemos apreciar por la evolución del número de puntos y la distancia al objeto, inicialmente se encuentra alejado, y se va acercando. Los errores se reducen cuanto más cerca se encuentra el objeto y aumentan cuando el número de puntos es bajo o existe alguna oclusión.

A continuación, en la siguiente tabla se muestran los valores medios de errores y el tiempo de ejecución:

CASO 1	Objeto 0
TP frames	22
TFNP [%]	37.14
ERMP – KF [%]	16.91
ERMP – PT [%]	7.79
RECM – KF [km/h]	2.84
RECM – PT [km/h]	1.43
Tiempo medio de ejecución [ms/frame]	40.81

TABLA 6.3-1: Resultados comparativos del caso de objeto estático.

Se observa que el error de seguimiento de velocidad cuando únicamente se evalúa un objeto con el sistema presentado se reduce considerablemente cuando se emplea el algoritmo de precision-tracking en relación con un simple filtro de Kalman.

6.3.1.2. Caso 1: Múltiples objetos estáticos

Para comprobar la robustez del método, en este caso se realizará el seguimiento de múltiples objetos estáticos, comparando la velocidad con el *groundtruth* del mismo modo que en el caso anterior.

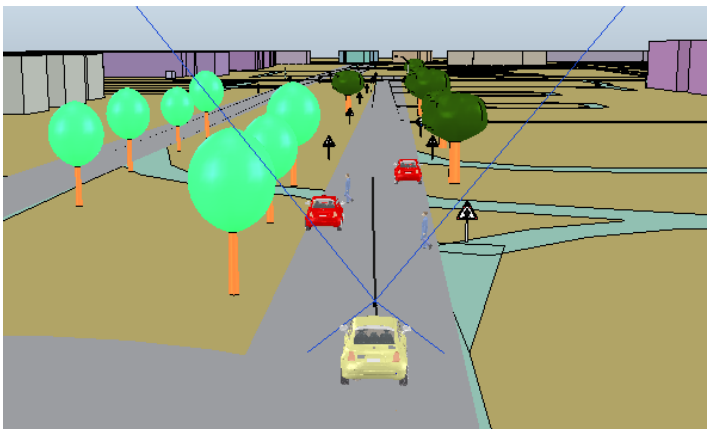

FIGURA 6.3-3: Escena de seguimiento caso múltiples objetos estáticos.

En esta ocasión se evaluarán 4 objetos (2 coches y 2 peatones) para observar el desempeño del método cuando se siguen varios objetos de forma simultánea. En las siguientes gráficas se muestra el error de velocidad para cada uno de ellos:

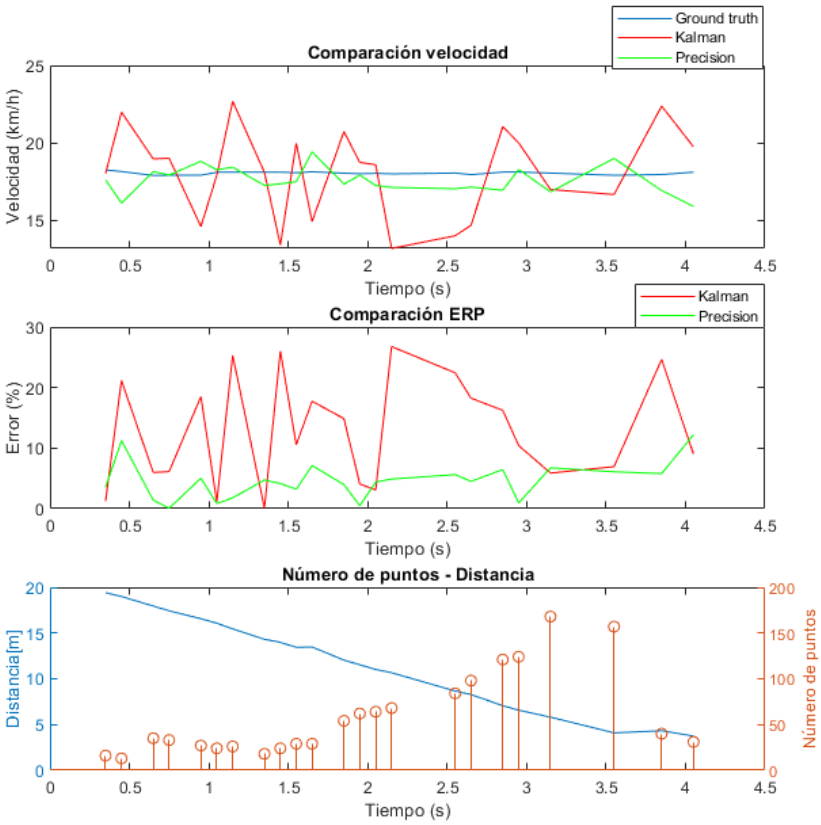

Figura 6.3-4: Gráfica comparativa de seguimiento de velocidad caso múltiples objetos estáticos (objeto0).

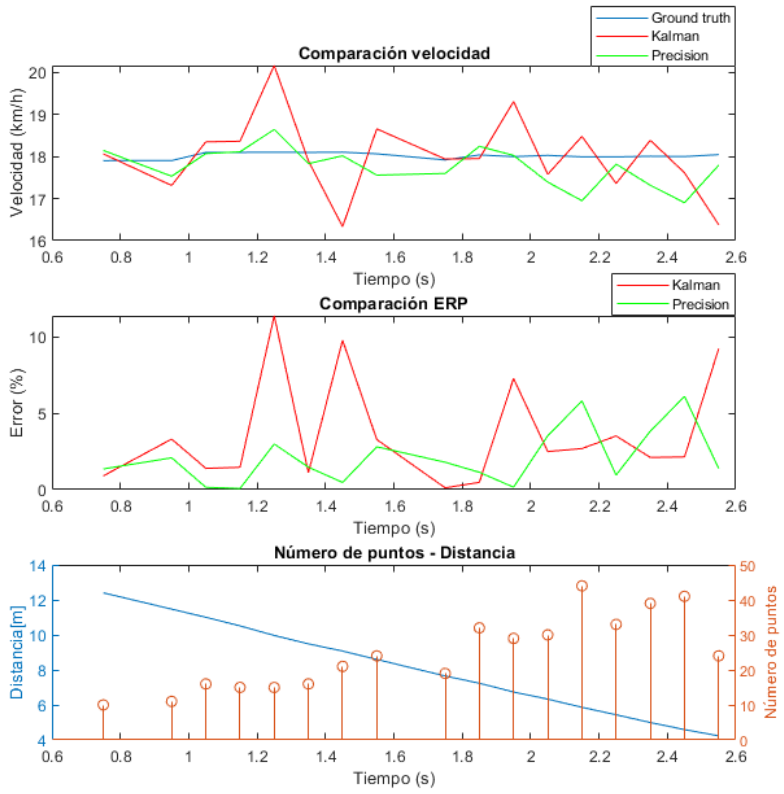

Figura 6.3-5: Gráfica comparativa de seguimiento de velocidad caso múltiples objetos estáticos (objeto1).

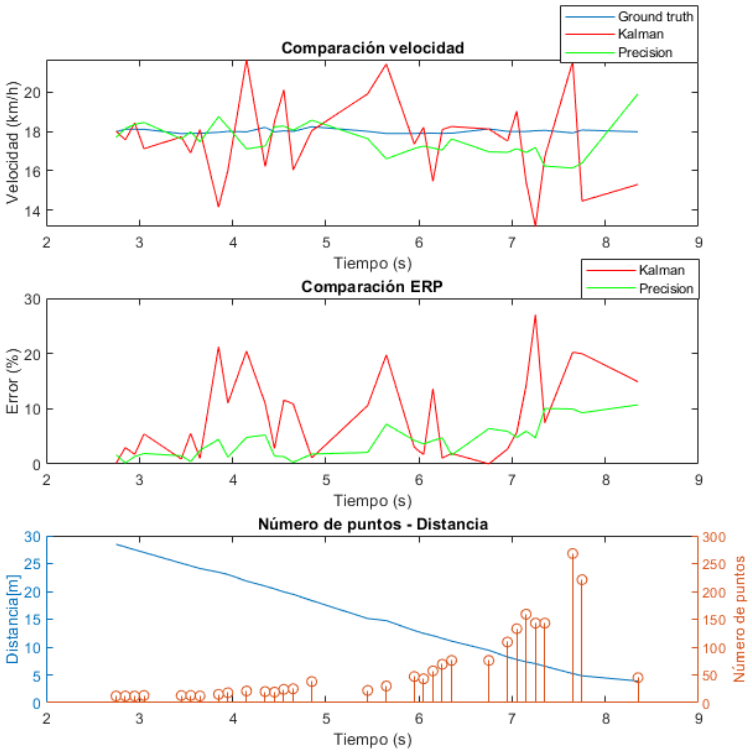

Figura 6.3-6: Gráfica comparativa de seguimiento de velocidad caso múltiples objetos estáticos (objeto2).

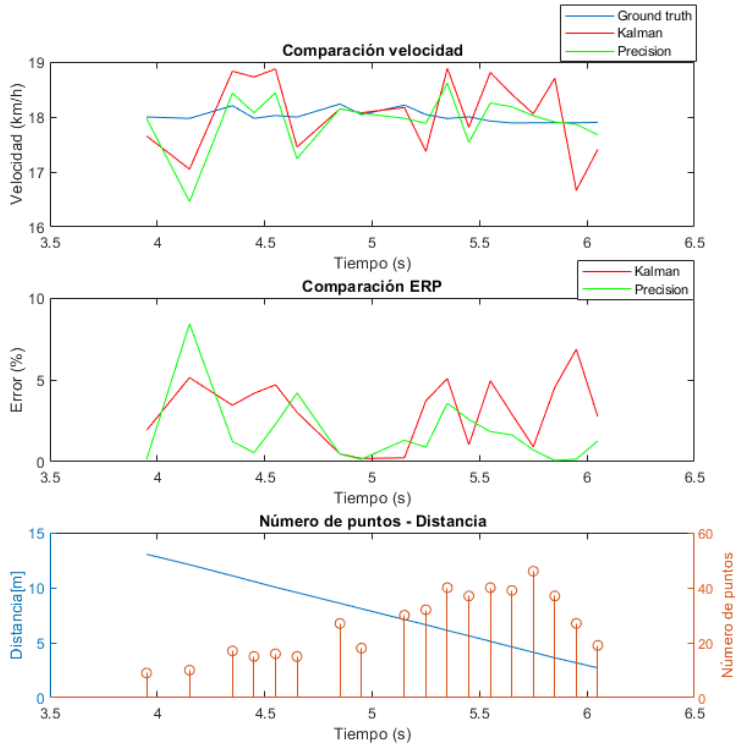

Figura 6.3-7: Gráfica comparativa de seguimiento de velocidad caso múltiples objetos estáticos (objeto3).

Como podemos ver la evolución del número de puntos en cada uno de los objetos determina el movimiento que sigue el propio vehículo acercándose a ellos hasta sobrepasarlos. Llama la atención los instantes en los que no existen puntos debido a errores de detección, donde el filtro de Kalman aumenta su error, mientras que el caso de precision-tracking se mantiene en un seguimiento más constante.

A continuación, en la siguiente tabla se muestra una comparativa de errores para cada uno de los objetos:

CASO 2	Objeto 0	Objeto 1	Objeto 2	Objeto 3
TP frames	23	17	31	18
TFNP [%]	43.90	10.52	42.59	10
ERMP – KF [%]	12.89	3.69	8.76	3.11
ERMP – PT [%]	4.57	2.12	4.06	1.75
RECM – KF [km/h]	2.8	0.9	2.1	0.66
RECM – PT [km/h]	0.99	0.49	0.91	0.48
Tiempo medio de ejecución [ms/frame]	44.4832			

TABLA 6.3-2: Resultados comparativos del caso de Seguimiento 2

Como podemos apreciar, la tendencia de error se mantiene en los 4 objetos presentando el filtro de Kalman errores superiores a precision-tracking. Los objetos que presentan una peor detección se ven reflejados en el aumento de la TFNP, suponiendo un mayor error.

6.3.1.3. Caso 2: Múltiples objetos en movimiento

Una vez comparados los errores con respecto a la velocidad real, se va a medir la evolución de la velocidad de varios objetos en movimiento considerando la velocidad relativa de desplazamiento:

$$\boxed{V_B = V_{BA} + V_A}$$

Donde:

- V_A: es la velocidad del propio vehículo.
- V_B: es la velocidad del objeto que se sigue.
- V_{BA}: es la velocidad del objeto que se sigue con respecto al propio vehículo.

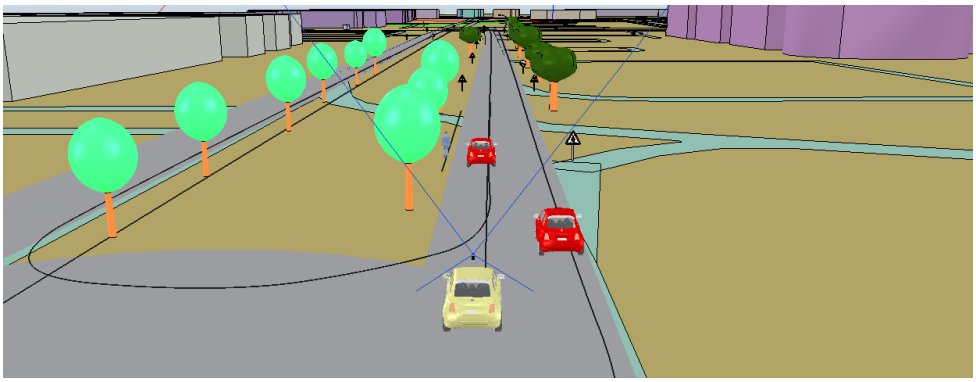

FIGURA 6.3-8: Escena de seguimiento caso múltiples objetos en movimiento.

En las siguientes gráficas podemos ver cómo varía la velocidad en cada uno de los métodos, así como su diferencia en valor real y absoluto:

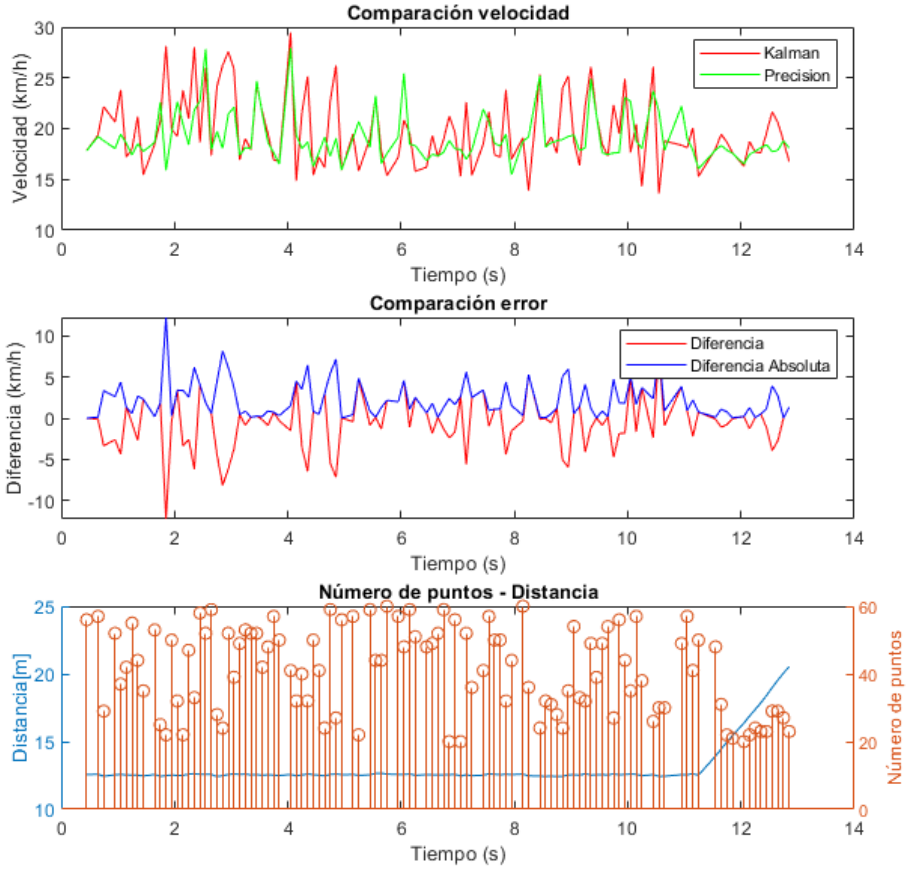

FIGURA 6.3-9: Gráfica de seguimiento de velocidad caso múltiples objetos en movimiento (objeto 0).

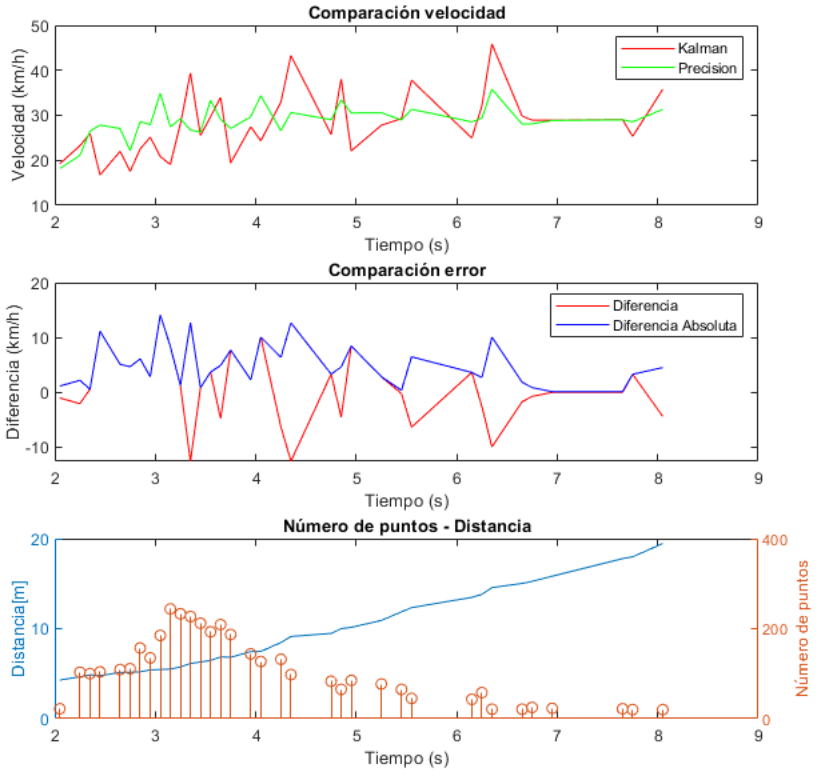

FIGURA 6.3-10: Gráfica de seguimiento de velocidad caso múltiples objetos en movimiento (objeto 1).

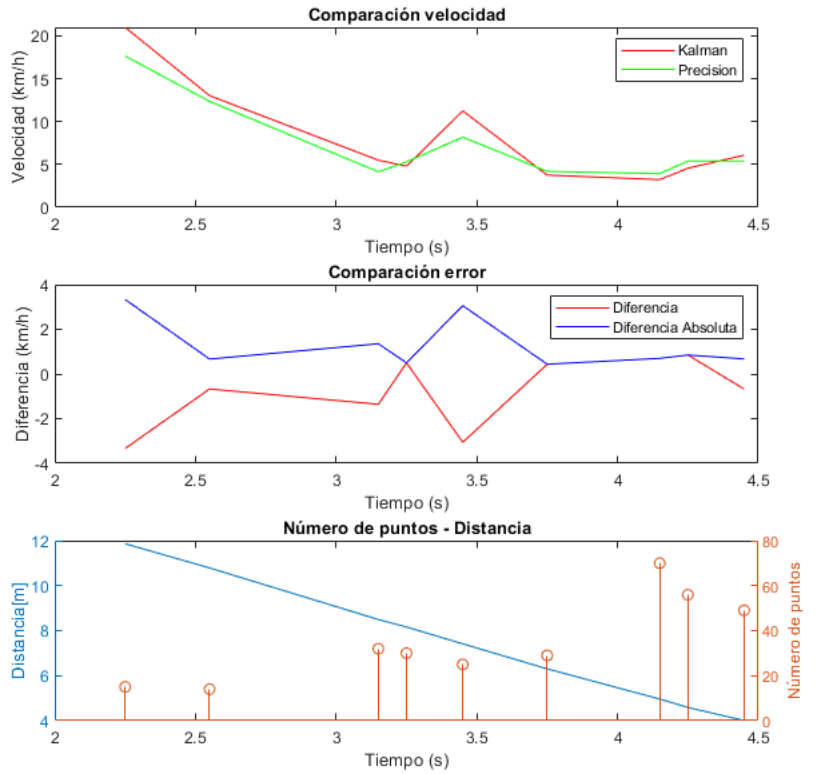

FIGURA 6.3-11: Gráfica de seguimiento de velocidad caso múltiples objetos en movimiento (objeto 2).

La evolución de la distancia y el número de puntos determina las trayectorias seguidas por los objetos: el objeto 0 se mantiene a una velocidad similar al propio vehículo, mientras que el objeto 1 realiza un adelantamiento, de ahí que inicialmente es cuando más cerca se encuentra, y el objeto 2 se va acercando hasta que es sobrepasado. Como podemos ver la detección de velocidad mediante Kalman presenta variaciones más bruscas y se mantiene a menudo con estimaciones superiores a las de precision-tracking.

A continuación, en la siguiente tabla se muestran una comparativa de errores para cada uno de los objetos:

CASO 3	Objeto 0	Objeto 1	Objeto 2
TP frames	108	35	9
TFNP [%]	11.47	38.59	52.63
Error Medio (PT-KF) [km/h]	-0.63	0.81	-0.73
Error Absoluto Medio (PT-KF) [km/h]	2.25	4.86	1.29
RECM (PT-KF) [km/h]	3.14	6.23	1.66
Velocidad Media Kalman [km/h]	19.79	27.93	7.39
Velocidad Media precisión-tracking [km/h]	19.16	28.75	8.13
Tiempo medio de ejecución [ms/frame]	53.92		

TABLA 6.3-3: Resultados comparativos del caso de Seguimiento 3.

Como se observa, la velocidad media estimada no difiere mucho entre los distintos tipos de seguimiento para cada objeto, sin embargo, debido a que el método de precisión-tracking presenta un menor error, podemos determinar que estará más cerca del valor de velocidad real.

6.3.2. Caso real

En este caso se registrarán los datos obtenidos del mismo bag en el que se analizó la detección. Como únicamente aparecerán vehículos aparcados a los lados de la carretera, la velocidad relativa detectada debería ser la misma a la que se desplaza nuestro vehículo. Este caso es el equivalente al de "Múltiples objetos en movimiento" probado en simulación:

FIGURA 6.3-12: Escena de seguimiento caso real.

En esta ocasión se evaluarán varios vehículos para observar el desempeño del método cuando se siguen varios objetos de forma simultánea. En las siguientes gráficas se muestra el error de velocidad de alguno de ellos:

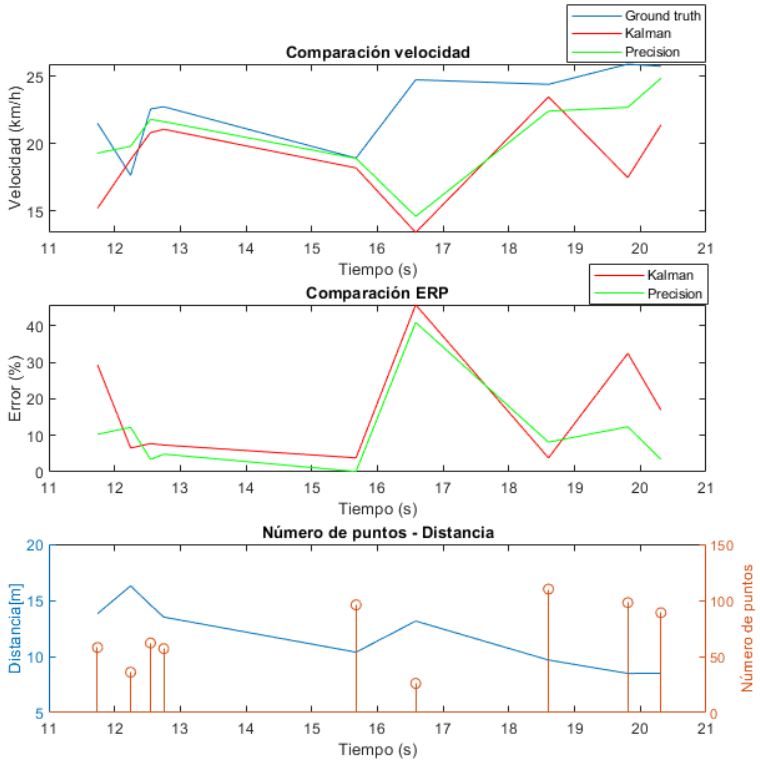

Figura 6.3-13: Gráfica comparativa de seguimiento de velocidad real (objeto 0).

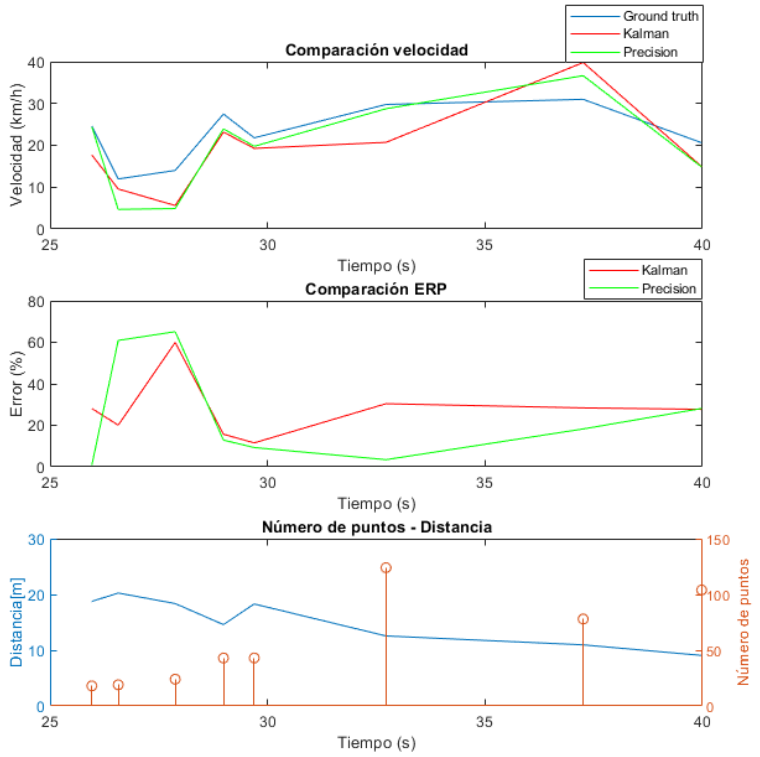

FIGURA 6.3-14: Gráfica comparativa de seguimiento de velocidad real (objeto 1).

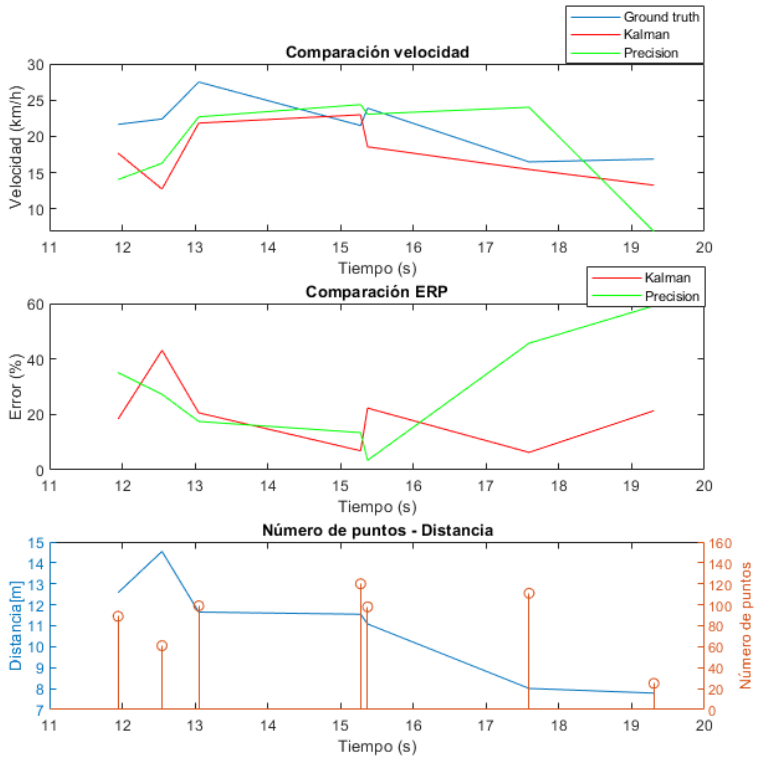

FIGURA 6.3-15: Gráfica comparativa de seguimiento de velocidad real (objeto 2).

A continuación, en la siguiente tabla se muestra una comparativa de errores para cada uno de los objetos:

CASO REAL	Objeto 0	Objeto 1	Objeto 2
TP frames	9	8	7
TFNP [%]	55	65.22	56.25
ERMP – KF [%]	17.09	27.74	19.83
ERMP – PT [%]	10.64	24.88	28.79
RECM – KF [km/h]	5.43	6.53	5.15
RECM – PT [km/h]	3.78	5.24	6.36
Tiempo medio de ejecución [ms/frame]	87.45		

TABLA 6.3-4: Resultados comparativos del caso de seguimiento real.

Como podemos apreciar, el seguimiento en el caso real es peor que en simulación, que se aprecia en la elevada tasa de falsos negativos. El error debido a precision-tracking suele ser más pequeño y mantenerse más constante que en el caso del filtro de Kalman, salvo en el último objeto expuesto.

Capítulo VII

CONCLUSIONES Y TRABAJOS FUTUROS

7.1. Conclusiones

En este trabajo se ha expuesto el desarrollo e implementación de un sistema completo de detección y seguimiento en un entorno urbano simulado con vehículos y peatones tomando como base la fusión de datos multisensoriales obtenidos por un LiDAR 3D y una cámara.

Debe considerarse que el enfoque planteado para realizar el seguimiento de los objetos depende de manera crucial de los resultados obtenidos durante la detección, ya que son los datos que emplea como entrada. Durante el proceso de detección debe lidiarse tanto con las incertidumbres de medida de los sensores como con las oclusiones causadas por cambios de punto de vista, que hace que láser no refleje algunos puntos correctamente, además de los errores que puedan surgir por una fusión de información sensorial no perfecta.

Uno de los puntos clave de los problemas de detección sufridos se encuentra en que se emplea un LiDAR 3D de 16 haces, que en comparación con otros del mercado de 32 o 64 haces, presenta menos campo de visión y resolución angular, sin embargo, también tiene un menor precio y requiere de un menor esfuerzo computacional, lo que hace que resulte atractivo el desarrollo de algoritmos para perfeccionar la detección.

Durante la segmentación existe un compromiso en la elección del número de puntos mínimo que debe considerarse para un clúster de objeto. Limitar el tamaño mínimo a un número pequeño de puntos tiene la ventaja de poder determinar la presencia de un objeto que se encuentra a mayor distancia, sin embargo, es más probable que se caiga en un error de detección, al contrario que si se aumenta el tamaño del clúster mínimo, que garantiza una detección más precisa, pero cuando el objeto se encuentra relativamente cercano.

Como era de esperar los resultados de seguimiento obtenidos mediante el algoritmo de precisión-tracking mejoran notablemente el seguimiento realizado por un filtro de Kalman, al incluir también la información de color. Estas mejoras se deben esencialmente al alineamiento sobre la nube de puntos en instantes sucesivos.

En las pruebas realizadas en el sistema real no se obtuvieron los resultados con la misma precisión que en la simulación, debido a las restricciones del mundo real que no se consideran en la simulación, lo que muestra que el sistema es susceptible de ser mejorado.

Además, los bajos tiempos de ejecución extraídos, a pesar de que para ser más concluyentes se necesitaría un mayor número de pruebas, pueden determinar la posibilidad de realizar un seguimiento en tiempo real.

Por tanto, puede concluirse que los objetivos planteados respecto al estudio e implementación para abordar la funcionalidad de un sistema de detección y seguimiento han sido completados.

7.2. Trabajos futuros

En vista de las conclusiones extraídas se plantea la posibilidad de realizar los siguientes trabajos futuros:

- Los casos de evaluación considerados reflejan una escena típica de desplazamiento de objetos en la misma dirección de avance del vehículo, facilitando la detección de objetos, por lo que en un análisis futuro de escenas más complejas sería necesario desarrollar un modelo de detección de orientación de los objetos.

- La fusión de datos sensoriales utiliza únicamente el campo de visión de la cámara, desechando el resto de información que queda a los laterales y por detrás del vehículo, por lo que un trabajo interesante para tener un mayor control del seguimiento en la escena completa podría ser detectar los objetos sólo con la información de los puntos del láser, y reforzar los resultados de detección con el color en caso de que estos objetos entrasen en el campo de visión.

- Sólo se han considerado la detección de los objetos por el tamaño de la nube de puntos detectada y por su color, pero podría resultar interesante el estudio de otros algoritmos de detección como VFH (Viewpoint Feature Histogram) que compara las similitudes geométricas de los clústeres, con el que probablemente se podrían conseguir mejorar los resultados.

- En este trabajo se emplea la información de una de las cámaras para obtener la información de color de la escena. A pesar de la limitación en el esfuerzo computacional en el procesamiento de imágenes estereoscópicas para realizar un sistema en tiempo real, podría plantearse el hecho de obtener los puntos de profundidad de la imagen estéreo para corregir los puntos obtenidos por el láser y así mejorar la fusión multisensorial.

- Teniendo en cuenta que el Velodyne gira a 10Hz, limita el tiempo de procesamiento entre frames a 100ms, sin embargo, en los resultados obtenidos el tiempo medio se ha mantenido por debajo de este valor por lo tanto sería interesante poder probarlo en un sistema de forma online.

- Además, debe considerarse que la programación del sistema únicamente se ha implementado en un hilo sin el uso de una GPU, lo que deja un margen de mejora del algoritmo pudiendo mantenerse las restricciones de tiempo real en el caso de plantearse un sistema multihilo paralelo.

PLANOS

En este capítulo el código principal que ha sido creado y modificado para poder llevar a cabo la implementación del sistema expuesto a lo largo del libro. Para una información más detallada, se invita al lector a acudir a las diferentes fuentes bibliográficas que han sido utilizadas.

PL.1. Código simulación

- **Lanzador.sh**

Se trata de un script que al ser lanzado permite la puesta en marcha de los launch, parámetros y nodos para la ejecución del sistema de detección y seguimiento de objetos simulado:

```
# Lanzador TFM: "Detección y seguimiento mediante precision-tracking"
# Autor: Samuel Pardo Alía

#! /bin/sh
echo "Script principal de lanzamiento para ejecutar el sistema"

source ~/catkin_ws/devel/setup.bash

Terminal1="roslaunch main_pkg navigation.launch"
Terminal2="roslaunch but_calibration_camera_velodyne coloring_vlp16.launch"
Terminal3="rosrun main_pkg tracking_object_vlp16"

$Terminal1 &
sleep 1
$Terminal2 &
sleep 1
$Terminal3 &
sleep 5
gnome-terminal --tab -e
rosparam set use_sim_time true
bash ~/V-REP_PRO_EDU_V3_4_0_Linux/vrep.sh
```

- **navigation.launch**

Archivo launch que ejecuta rviz cargando el modelo del coche y de todos sus sensores y parámetros:

```
<?xml version="1.0"?>
<launch>
    <param name="use_sim_time" value="true" />

    <node pkg="main_pkg" name="local_transform_velodyne" type="local_transform_velodyne"/>

    <param name="axis_linear"          value="2"              type="int" />
    <param name="axis_angular"         value="0"              type="int" />
    <param name="axis_brake"           value="1"              type="int" />

    <param name="button_reverse"       value="3"              type="int" />

    <param name="scale_linear"         value="80"             type="double" />
    <param name="scale_angular"        value="-9.42"          type="double" />
    <param name="scale_brake"          value="1"              type="double" />

    <param name="offset_reverse"       value="0"              type="double" />

    <node name="GUI_visual" pkg="rviz" type="rviz" args="-d $(find main_pkg)/rviz/Velodyne_cam.rviz"/>
    <param name="robot_description" textfile="$(find main_pkg)/urdf/car.urdf" />

</launch>
```

- **local_transform_velodyne.cpp**

Nodo cargado por *navigation.launch* que permite la transformación de las coordenadas globales de los sensores a coordenadas locales:

```
#include<ros/ros.h>
#include<sensor_msgs/PointCloud2.h>
#include<pcl_ros/transforms.h>
#include<tf/transform_listener.h>
#include<iostream>

class PointTransformation{
public:
 PointTransformation();

private:
 void callBack(const sensor_msgs::PointCloud2::ConstPtr& msg);
 ros::NodeHandle n;
 ros::Publisher pub;
 ros::Subscriber sub;
 tf::TransformListener listener;
};

PointTransformation::PointTransformation()
{
   pub = n.advertise<sensor_msgs::PointCloud2>("/velodyne_points_local",1);
   sub = n.subscribe<sensor_msgs::PointCloud2>("/velodyne_points", 1, &PointTransformation::callBack,
this);
}

void PointTransformation::callBack(const sensor_msgs::PointCloud2::ConstPtr& msg)
{

   sensor_msgs::PointCloud2 local_PC; //this will contain the transformed point cloud

   /*This tries to transform the point cloud according to the next available Transform*/
   listener.waitForTransform("/odom","/base_laser",ros::Time::now(), ros::Duration(1.0));
 //wait for the next tf update
```

```
  pcl_ros::transformPointCloud("/base_laser", *msg, local_PC, listener); //transform the global point
cloud to the "/base_laser" frame
  /**/
  pub.publish(local_PC);
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "local_transform_velodyne");
  PointTransformation Transformation;
  ros::spin();
  return(0);
}
```

- **coloring_vlp16.launch**

Archivo launch que carga los parámetros de calibración y realiza la fusión de la información del LiDAR y la cámara:

```
<?xml version="1.0"?>
<launch>
  <rosparam command="load" file="$(find but_calibration_camera_velodyne)/conf/calibration_vlp16.yaml" />
  <rosparam command="load" file="$(find but_calibration_camera_velodyne)/conf/coloring_vlp16.yaml" />
  <node pkg="but_calibration_camera_velodyne" type="coloring_vlp16" name="coloring_vlp16" output="screen">
  </node>
</launch>
```

- **coloring-node_vlp16.cpp**

Nodo cargado por *coloring_vlp16.launch* que realiza la fusión multisensorial publicando un mensaje con los puntos del LiDAR a color:

```
/* manual_calibration.cpp*
 *  Created on: 27.2.2014  Author: ivelas */

                                        [Includes]
...

string CAMERA_FRAME_TOPIC;
string CAMERA_INFO_TOPIC;
string VELODYNE_TOPIC;
string VELODYNE_COLOR_TOPIC;

ros::Publisher pub;
cv::Mat projection_matrix;

cv::Mat frame_rgb;
vector<float> DoF;

void cameraInfoCallback(const sensor_msgs::CameraInfoConstPtr& msg)
{
 float p[12];
 float *pp = p;
 for (boost::array<double, 12ul>::const_iterator i = msg->P.begin(); i != msg->P.end(); i++)
 {
  *pp = (float)(*i);
  pp++;
 }
 cv::Mat(3, 4, CV_32FC1, &p).copyTo(projection_matrix);
}
```

```
void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
  cv_bridge::CvImagePtr cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
  frame_rgb = cv_ptr->image;
}

void pointCloudCallback(const sensor_msgs::PointCloud2ConstPtr& msg)
{
   if (frame_rgb.data == NULL) // if no rgb frame for coloring:
       return;
  PointCloud<Velodyne::Point> pc;
  fromROSMsg(*msg, pc);

  // x := x, y := -z, z := y,
  Velodyne::Velodyne pointcloud = Velodyne::Velodyne(pc).transform(0, 0, 0, M_PI / 2, 0, 0);
  Image::Image img(frame_rgb);
  Velodyne::Velodyne transformed = pointcloud.transform(DoF);
  PointCloud<Velodyne::Point> visible_points;
  transformed.project(projection_matrix, Rect(0, 0, frame_rgb.cols, frame_rgb.rows), &visible_points);

  Velodyne::Velodyne visible_scan(visible_points);
  PointCloud<PointXYZRGB> color_cloud = visible_scan.colour(frame_rgb, projection_matrix);

  // reverse axix switching:
  Eigen::Affine3f transf = getTransformation(0, 0, 0, -M_PI / 2, -M_PI / 2, -M_PI / 2);
  transformPointCloud(color_cloud, color_cloud, transf);

  sensor_msgs::PointCloud2 color_cloud2;
  toROSMsg(color_cloud, color_cloud2);
  color_cloud2.header = msg->header;
  pub.publish(color_cloud2);
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "coloring_node");
  ros::NodeHandle n;
  n.getParam("/but_calibration_camera_velodyne/camera_frame_topic", CAMERA_FRAME_TOPIC);
  n.getParam("/but_calibration_camera_velodyne/camera_info_topic", CAMERA_INFO_TOPIC);
  n.getParam("/but_calibration_camera_velodyne/velodyne_topic", VELODYNE_TOPIC);
  n.getParam("/but_calibration_camera_velodyne/velodyne_color_topic", VELODYNE_COLOR_TOPIC);
  n.getParam("/but_calibration_camera_velodyne/6DoF", DoF);

  pub = n.advertise<sensor_msgs::PointCloud2>(VELODYNE_COLOR_TOPIC, 1);

  // Subscribe input camera image
  image_transport::ImageTransport it(n);
  image_transport::Subscriber sub = it.subscribe(CAMERA_FRAME_TOPIC, 10, imageCallback);

  ros::Subscriber info_sub = n.subscribe(CAMERA_INFO_TOPIC, 10, cameraInfoCallback);
  ros::Subscriber pc_sub = n.subscribe<sensor_msgs::PointCloud2>(VELODYNE_TOPIC, 1,
                                  pointCloudCallback);

  ros::spin();

  return EXIT_SUCCESS;
}
```

PL.2. Código real

- **Lanzador_real.sh**

Se trata de un script que al ser lanzado permite la puesta en marcha de los launch, parámetros y nodos para la ejecución del sistema de detección y seguimiento de objetos real:

```
# Lanzador TFM: "Detección y seguimiento mediante precision-tracking"
# Autor: Samuel Pardo Alía

#! /bin/sh
echo "Script principal de lanzamiento para ejecutar el sistema real"

source ~/catkin_ws/devel/setup.bash

Terminal1="roslaunch smart_eld_car navigation_real.launch"
Terminal2="roslaunch velo2cam_calibration pcl_coloring_real.launch"
Terminal3="roslaunch smart_eld_car genera_tf_real.launch"
Terminal4="rosrun vehicle_target tracking_object_real"

$Terminal1 &
sleep 1
$Terminal2 &
sleep 1
$Terminal3 &
sleep 1
$Terminal4 &
```

- **local_transform_camera.cpp**

Nodo cargado por *navigation.launch* que permite la transformación de las coordenadas de la cámara al las coordendas del velodyne:

```
#include<ros/ros.h>
#include<sensor_msgs/PointCloud2.h>
#include<pcl_ros/transforms.h>
#include<tf/transform_listener.h>
#include<iostream>

class PointTransformation{
public:
 PointTransformation();
private:
 void callBack(const sensor_msgs::PointCloud2::ConstPtr& msg);
 ros::NodeHandle n;
 ros::Publisher pub;
 ros::Subscriber sub;
 tf::TransformListener listener;
};
PointTransformation::PointTransformation()
{
   pub = n.advertise<sensor_msgs::PointCloud2>("/velodyne_coloured_base_link",1);
   sub = n.subscribe<sensor_msgs::PointCloud2>("/velodyne_coloured_points", 1,
&PointTransformation::callBack, this);
}
```

```
void PointTransformation::callBack(const sensor_msgs::PointCloud2::ConstPtr& msg)
{
  sensor_msgs::PointCloud2 local_PC; //this will contain the transformed point cloud

  /*This tries to transform the point cloud according to the next available Transform*/
  listener.waitForTransform("/stereo_camera_LR","/velodyne",ros::Time::now(), ros::Duration(1.0));
//wait for the next tf update
  pcl_ros::transformPointCloud("/velodyne", *msg, local_PC, listener); //transform the global point
cloud to the "/base_laser" frame

  pub.publish(local_PC);

}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "local_transform_camerAa");
  PointTransformation Transformation;
  ros::spin();
  return(0);
}
```

- **pcl_coloring_real.launch**

Archivo launch que carga los parámetros de calibración y realiza la fusión de la información del LiDAR y la cámara:

```
<?xml version="1.0"?>
<launch>

        <node pkg="velo2cam_calibration" type="pcl_coloring_real" name="pcl_coloring_real" output="screen"

                <remap from="pcl_coloring_real/pointcloud" to="/velodyne_points"/>
                <remap from="pcl_coloring_real/camera_info" to="/camera/stereo_camera_LR/left/camera_info"
<!--            <remap from="pcl_coloring_real/image" to="/camera/stereo_camera_LR/left/image_color"/>-->
                <remap from="pcl_coloring_real/image" to="/segmentation_output"/>
                <remap from="/pcl_coloring_real/velodyne_coloured" to="/velodyne_coloured_points"/>

                <param name="source_frame" value="velodyne"/>
                <param name="target_frame" value="stereo_camera_LR"/>
        </node>

</launch>
```

- **pcl_coloring_real.cpp**

Nodo cargado por *pcl_coloring_real.launch* que realiza la fusión multisensorial publicando un mensaje con los puntos del LiDAR a color:

```
[Includes]...

typedef pcl::PointCloud<pcl::PointXYZRGB> PointCloudXYZRGB;
ros::Publisher pcl_pub;
string target_frame, source_frame;
void callback(const PointCloud2::ConstPtr& pcl_msg, const CameraInfoConstPtr& cinfo_msg, const
ImageConstPtr& image_msg){
  cv_bridge::CvImageConstPtr cv_img_ptr;
  try{
    cv_img_ptr = cv_bridge::toCvShare(image_msg);
  }catch (cv_bridge::Exception& e){
    ROS_ERROR("cv_bridge exception: %s", e.what());
    return;
  }
```

```
  cv::Mat image(cv_img_ptr->image.rows, cv_img_ptr->image.cols, cv_img_ptr->image.type());
  image = cv_bridge::toCvShare(image_msg)->image;

  image_geometry::PinholeCameraModel cam_model_;
  cam_model_.fromCameraInfo(cinfo_msg);

  pcl::PointCloud<pcl::PointXYZ>::Ptr pcl_cloud(new pcl::PointCloud<pcl::PointXYZ>); // From ROS Msg
  pcl::PointCloud<pcl::PointXYZ>::Ptr trans_cloud(new pcl::PointCloud<pcl::PointXYZ>); // After
transformation
  PointCloudXYZRGB::Ptr coloured = PointCloudXYZRGB::Ptr(new PointCloudXYZRGB); // For coloring
purposes
  fromROSMsg(*pcl_msg, *pcl_cloud);

  tf::TransformListener listener;
  tf::StampedTransform transform;
  try{
   listener.waitForTransform(target_frame.c_str(), source_frame.c_str(), ros::Time(0),
ros::Duration(20.0));
   listener.lookupTransform (target_frame.c_str(), source_frame.c_str(), ros::Time(0), transform);
  }catch (tf::TransformException& ex) {
   ROS_WARN("[draw_frames] TF exception:\n%s", ex.what());
   return;
  }
  pcl_ros::transformPointCloud (*pcl_cloud, *trans_cloud, transform);
  trans_cloud->header.frame_id = target_frame;
  pcl::copyPointCloud(*trans_cloud, *coloured);

for (pcl::PointCloud<pcl::PointXYZRGB>::iterator pt = coloured->points.begin(); pt <
coloured->points.end(); pt++)
  {
                        cv::Point3d pt_cv((*pt).x, (*pt).y, (*pt).z);
            cv::Point2d uv;
            uv = cam_model_.project3dToPixel(pt_cv);
            if(uv.x>0 && uv.x < image.cols && uv.y > 0 && uv.y < image.rows &&(*pt).z>=0){
    // Copy colour to laser  simulation VREP
     (*pt).r = image.at<cv::Vec3b>(uv)[0];
     (*pt).g = image.at<cv::Vec3b>(uv)[1];
     (*pt).b = image.at<cv::Vec3b>(uv)[2];
                        }
             else  // color black cloud to white/grey
                        {
            (*pt).r = 20.0;
     (*pt).g = 20.0;
     (*pt).b = 20.0;
                        }
  }

int main(int argc, char **argv){
  ros::init(argc, argv, "pcl_coloring_real");
  ros::NodeHandle nh_("~"); // LOCAL
  nh_.param<std::string>("target_frame", target_frame, "/stereo_camera_LR");
  nh_.param<std::string>("source_frame", source_frame, "/velodyne");
  // Subscribers
  message_filters::Subscriber<PointCloud2> pc_sub(nh_, "pointcloud", 1);
  message_filters::Subscriber<CameraInfo> cinfo_sub(nh_, "camera_info", 1);
  message_filters::Subscriber<Image> image_sub(nh_, "image", 1);
  pcl_pub = nh_.advertise<PointCloud2> ("velodyne_coloured", 1);
  typedef message_filters::sync_policies::ApproximateTime<PointCloud2, CameraInfo, Image>
MySyncPolicy;
  message_filters::Synchronizer<MySyncPolicy> sync(MySyncPolicy(10), pc_sub, cinfo_sub, image_sub);
  sync.registerCallback(boost::bind(&callback, _1, _2, _3));

  ros::spin();
  return 0;
}
```

PL.3. Nodo principal

- **tracking_object_vlp16.cpp / tracking_object_real.cpp**

Nodo principal del sistema que aúna el proceso de detección y seguimiento:

```
[Includes] . . .

//////// ESTRUCTURAS ////////////////////////
struct Objeto
{
        Eigen::Vector3f centroid;
        double r; double g; double b;
        float x_min; float y_min; float z_min;float x_max; float y_max; float z_max;
        float w; float h; float d;
        pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud;
        const char *tipo;
}objeto, car, pedestrian;

typedef struct Kalman_State
{
        double x; double y; double z;double w; double h; double d;
        double time;
}estado_kalman;

typedef struct Array_Kalman_States
{
        double x[10]; double y[10]; double z[10]; double w[10]; double h[10]; double d[10];
        double time[10];
}array_estados_kalman;
struct Precision_Tracker
{
        Eigen::Vector3f velocity_estimated;
}precisiontracker;

//DEFINICIONES
#define LIDAR_HEIGHT 1.73 // Altura a la que se encuentra el LIDAR
//Dimensiones
#define CAR_LENGTH 3.5
#define CAR_WIDTH 1.65
#define CAR_HEIGHT 1.5
#define PEDESTRIAN_LENGTH 0.5
#define PEDESTRIAN_WIDTH 0.5
#define PEDESTRIAN_HEIGHT 1.8

////// VARIABLES ///////////////////////////////

// DECLARAR LOS SUBSCRIBER Y PUBLISHER
ros::Subscriber lidar_sub_, odom_sub_;
ros::Publisher obstacle_pub_;

// DECLARAR EL TIMER
ros::Timer timer_;

int numero_objetos_inicialtotal=0;
int coche=0;
int peaton=0;

boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer (new pcl::visualization::PCLVisualizer ("3D
Viewer"));
pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_frame (new pcl::PointCloud<pcl::PointXYZRGB>);
```

```
std::vector<Objeto> objetos, car_vector, pedestrian_vector;
std::vector<Array_Kalman_States> kfs_array;
std::vector<cv::KalmanFilter> kfs;
std::vector<int> kfsTime;// Vector for Kalman filters time of life

Eigen::Vector3f centroid_pt=Eigen::Vector3f::Zero();
Eigen::Vector3f kf_vel=Eigen::Vector3f::Zero();Eigen::Vector3f gt_vel=Eigen::Vector3f::Zero();
Eigen::Vector3f kf_rvel=Eigen::Vector3f::Zero();Eigen::Vector3f pt_rvel=Eigen::Vector3f::Zero();

float velocidad_kalman=0;
float velocidad_real=0;
float velocidad_precision=0;
float vel_rel_kf=0;
float vel_rel_pt=0;
float frame_time=0;
int iterador;
float A, B, C, D; //Ecuación del plano del suelo (ax + by + cz + d = 0 )

//-------------------PrecisionTracker-------------------------------------------------------------------------------------------
std::vector<Precision_Tracker> tracker_vector;
std::vector<precision_tracking::Tracker> trackers;
Eigen::Vector3f estimated_velocity;
precision_tracking::Params params;
double sensor_horizontal_resolution, sensor_vertical_resolution;
//-----------------------------------------------------------------------------------------------------------------------------

////////////////////////////  DECLARACIÓN FUNCIONES  ////////////////////
// Initializes a Kalman filter and puts the initial values in the matrices used by the algorithm
cv::KalmanFilter initKalman(float x,float y,float z,float w,float h,float d,float sigmaR1,float
sigmaR2,float sigmaR3,float sigmaQ1,float sigmaQ2,float sigmaQ3,float sigmaP)
{
        cv::KalmanFilter kf(12,6,0);
        //Matriz de transición (A)
        kf.transitionMatrix = (cv::Mat_<float>(12,12) <<        1,0,0,0,0,0,1,0,0,0,0,0,
                                                                0,1,0,0,0,0,0,1,0,0,0,0,
                                                                0,0,1,0,0,0,0,0,1,0,0,0,
                                                                0,0,0,1,0,0,0,0,0,1,0,0,
                                                                0,0,0,0,1,0,0,0,0,0,1,0,
                                                                0,0,0,0,0,1,0,0,0,0,0,1,
                                                                0,0,0,0,0,0,1,0,0,0,0,0,
                                                                0,0,0,0,0,0,0,1,0,0,0,0,
                                                                0,0,0,0,0,0,0,0,1,0,0,0,
                                                                0,0,0,0,0,0,0,0,0,1,0,0,
                                                                0,0,0,0,0,0,0,0,0,0,1,0,
                                                                0,0,0,0,0,0,0,0,0,0,0,1);
        //Matriz de medida (H)
        kf.measurementMatrix = (cv::Mat_<float>(6,12) <<        1,0,0,0,0,0,0,0,0,0,0,0,
                                                                0,1,0,0,0,0,0,0,0,0,0,0,
                                                                0,0,1,0,0,0,0,0,0,0,0,0,
                                                                0,0,0,1,0,0,0,0,0,0,0,0,
                                                                0,0,0,0,1,0,0,0,0,0,0,0,
                                                                0,0,0,0,0,1,0,0,0,0,0,0);
        //Matriz de covarianza del ruido del proceso (Q)
        kf.processNoiseCov = (cv::Mat_<float>(12,12) <<         sigmaQ1,0,0,0,0,0,0,0,0,0,0,0,
                                                                0,sigmaQ1,0,0,0,0,0,0,0,0,0,0,
                                                                0,0,sigmaQ2,0,0,0,0,0,0,0,0,0,
                                                                0,0,0,sigmaQ2,0,0,0,0,0,0,0,0,
                                                                0,0,0,0,sigmaQ3,0,0,0,0,0,0,0,
                                                                0,0,0,0,0,sigmaQ3,0,0,0,0,0,0,
                                                                0,0,0,0,0,0,sigmaQ1,0,0,0,0,0,
                                                                0,0,0,0,0,0,0,sigmaQ1,0,0,0,0,
                                                                0,0,0,0,0,0,0,0,sigmaQ2,0,0,0,
                                                                0,0,0,0,0,0,0,0,0,sigmaQ2,0,0,
                                                                0,0,0,0,0,0,0,0,0,0,sigmaQ3,0,
                                                                0,0,0,0,0,0,0,0,0,0,0,sigmaQ3);
```

```
	////Matriz de covarianza del ruido de la medida (R)
	kf.measurementNoiseCov = (cv::Mat_<float>(6,6) <<	sigmaR1,0,0,0,0,0,
							0,sigmaR1,0,0,0,0,
							0,0,sigmaR2,0,0,0,
							0,0,0,sigmaR2,0,0,
							0,0,0,0,sigmaR3,0,
							0,0,0,0,0,sigmaR3);

	// Matriz de covarianza de estimación de error a posteriori (P(k)): P(k)=(I-K(k)*H)*P'(k)
	kf.errorCovPost = (cv::Mat_<float>(12,12) <<		sigmaP,0,0,0,0,0,0,0,0,0,0,0,
							0,sigmaP,0,0,0,0,0,0,0,0,0,0,
							0,0,sigmaP,0,0,0,0,0,0,0,0,0,
							0,0,0,sigmaP,0,0,0,0,0,0,0,0,
							0,0,0,0,sigmaP,0,0,0,0,0,0,0,
							0,0,0,0,0,sigmaP,0,0,0,0,0,0,
							0,0,0,0,0,0,sigmaP,0,0,0,0,0,
							0,0,0,0,0,0,0,sigmaP,0,0,0,0,
							0,0,0,0,0,0,0,0,sigmaP,0,0,0,
							0,0,0,0,0,0,0,0,0,sigmaP,0,0,
							0,0,0,0,0,0,0,0,0,0,sigmaP,0,
							0,0,0,0,0,0,0,0,0,0,0,sigmaP);

	// Estado corregido (x(k)): x(k)=x'(k)+K(k)*(z(k)-H*x'(k))
	kf.statePost.at<float>(0) = x; kf.statePost.at<float>(1) = y; kf.statePost.at<float>(2) = z;
	kf.statePost.at<float>(3) = w;kf.statePost.at<float>(4) = h; kf.statePost.at<float>(5) = d;
	kf.statePost.at<float>(6) = 0; kf.statePost.at<float>(7) = 0; kf.statePost.at<float>(8) = 0;
	kf.statePost.at<float>(9) = 0; kf.statePost.at<float>(10) = 0; kf.statePost.at<float>(11) = 0;
	//Estado predicho (x'(k)): x(k)=A*x(k-1)
	kf.statePre.at<float>(0) = x; kf.statePre.at<float>(1) = y; kf.statePre.at<float>(2) = z;
	kf.statePre.at<float>(3) = w; kf.statePre.at<float>(4) = h; kf.statePre.at<float>(5) = d;

	return kf;
}

// Updates Kalman filter values, recalculates its prediction and corrects the measurement
void updateKalman(cv::KalmanFilter &kf, float x, float y, float z, float w, float h, float d, bool
useMeasurement)
{
	cv::Mat measurement(6,1,CV_32FC1);
	cv::Mat prediction = kf.predict();

	if (useMeasurement == false)
	{
		measurement.at<float>(0) = kf.statePre.at<float>(0);
		measurement.at<float>(1) = kf.statePre.at<float>(1);
		measurement.at<float>(2) = kf.statePre.at<float>(2);
		measurement.at<float>(3) = kf.statePre.at<float>(3);
		measurement.at<float>(4) = kf.statePre.at<float>(4);
		measurement.at<float>(5) = kf.statePre.at<float>(5);
	}else{
		measurement.at<float>(0) = x; measurement.at<float>(1) = y;
		measurement.at<float>(2) = z; measurement.at<float>(3) = w;
		measurement.at<float>(4) = h; measurement.at<float>(5) = d;
	}
	kf.correct(measurement); //Actualiza el estado predicho con la medida
}

Kalman_State getKalmanPrediction(cv::KalmanFilter kf) // Returns  values of Kalman filter prediction
{
	Kalman_State point_k;
	point_k.x=kf.statePre.at<float>(0); point_k.y=kf.statePre.at<float>(1);
	point_k.z=kf.statePre.at<float>(2); point_k.w=kf.statePre.at<float>(3);
	point_k.h=kf.statePre.at<float>(4); point_k.d=kf.statePre.at<float>(5);
	return point_k;
}
```

```
// Updates shift values for clustering, shift
void updateKalman_shift(int n_filtro,float x, float y, float z, float w, float h, float d, double tiempo)
{
	for (int pos_filtro=9;pos_filtro>0;pos_filtro--) //Desplazan los puntos y elimina el último
	{
		kfs_array[n_filtro].x[pos_filtro]=kfs_array[n_filtro].x[pos_filtro-1];
		kfs_array[n_filtro].y[pos_filtro]=kfs_array[n_filtro].y[pos_filtro-1];
		kfs_array[n_filtro].z[pos_filtro]=kfs_array[n_filtro].z[pos_filtro-1];
		kfs_array[n_filtro].w[pos_filtro]=kfs_array[n_filtro].w[pos_filtro-1];
		kfs_array[n_filtro].h[pos_filtro]=kfs_array[n_filtro].h[pos_filtro-1];
		kfs_array[n_filtro].d[pos_filtro]=kfs_array[n_filtro].d[pos_filtro-1];
		kfs_array[n_filtro].time[pos_filtro]=kfs_array[n_filtro].time[pos_filtro-1];
	}
	kfs_array[n_filtro].x[0]=x; kfs_array[n_filtro].y[0]=y; kfs_array[n_filtro].z[0]=z;
	kfs_array[n_filtro].w[0]=w; kfs_array[n_filtro].h[0]=h; kfs_array[n_filtro].d[0]=d;
	kfs_array[n_filtro].time[0]=tiempo;

	float kf_dis_x= kfs_array[n_filtro].x[0]-kfs_array[n_filtro].x[1];
	kf_vel(0)= kf_dis_x/(kfs_array[n_filtro].time[0]-kfs_array[n_filtro].time[1]);
	float kf_dis_y= (kfs_array[n_filtro].y[0]-(kfs_array[n_filtro].h[0]/2))-(kfs_array[n_filtro].y[1]-
		(kfs_array[n_filtro].h[1]/2));
	kf_vel(1)= kf_dis_y/(kfs_array[n_filtro].time[0]-kfs_array[n_filtro].time[1]);
	float kf_dis_z= kfs_array[n_filtro].z[0]-kfs_array[n_filtro].z[1];
	kf_vel(2)= kf_dis_z/(kfs_array[n_filtro].time[0]-kfs_array[n_filtro].time[1]);
}
void segmentation_filter (pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_filtered, float rMax, float rMin,
	float gMax, float gMin, float bMax, float bMin,  double r, double g, double b, int var, float
	tolerance, int min_cluster, int max_cluster)
{
	pcl::ConditionAnd<pcl::PointXYZRGB>::Ptr color_cond (new
		pcl::ConditionAnd<pcl::PointXYZRGB> ());
	color_cond->addComparison (pcl::PackedRGBComparison<pcl::PointXYZRGB>::Ptr (new
		pcl::PackedRGBComparison<pcl::PointXYZRGB> ("r", pcl::ComparisonOps::LT, rMax)));
	color_cond->addComparison (pcl::PackedRGBComparison<pcl::PointXYZRGB>::Ptr (new
		pcl::PackedRGBComparison<pcl::PointXYZRGB> ("r", pcl::ComparisonOps::GT, rMin)));
	color_cond->addComparison (pcl::PackedRGBComparison<pcl::PointXYZRGB>::Ptr (new
		pcl::PackedRGBComparison<pcl::PointXYZRGB> ("g", pcl::ComparisonOps::LT, gMax)));
	color_cond->addComparison (pcl::PackedRGBComparison<pcl::PointXYZRGB>::Ptr (new
		pcl::PackedRGBComparison<pcl::PointXYZRGB> ("g", pcl::ComparisonOps::GT, gMin)));
	color_cond->addComparison (pcl::PackedRGBComparison<pcl::PointXYZRGB>::Ptr (new
		pcl::PackedRGBComparison<pcl::PointXYZRGB> ("b", pcl::ComparisonOps::LT, bMax)));
	color_cond->addComparison (pcl::PackedRGBComparison<pcl::PointXYZRGB>::Ptr (new
		pcl::PackedRGBComparison<pcl::PointXYZRGB> ("b", pcl::ComparisonOps::GT, bMin)));

	// build the filter
	pcl::ConditionalRemoval<pcl::PointXYZRGB> condrem (color_cond);
	condrem.setInputCloud (cloud_filtered);
	condrem.setKeepOrganized(false);
	condrem.filter (*cloud_filtered); // apply filter

	//EXTRACCIÓN DE OBJECT
	if (cloud_filtered->points.size () >= min_cluster) //KDTREE necesita una nube con puntos
	{
		// Creating the KdTree object for the search method of the extraction
		pcl::search::KdTree<pcl::PointXYZRGB>::Ptr tree (new
			pcl::search::KdTree<pcl::PointXYZRGB>);
		tree->setInputCloud (cloud_filtered);
		std::vector<pcl::PointIndices> cluster_indices;
		pcl::EuclideanClusterExtraction<pcl::PointXYZRGB> ec;
		ec.setClusterTolerance (tolerance); //2 0.5
		ec.setMinClusterSize (min_cluster); //50
		ec.setMaxClusterSize (max_cluster); //25000
		ec.setSearchMethod (tree);
		ec.setInputCloud (cloud_filtered);
		ec.extract (cluster_indices);
```

```
//CALCULAR EL NUMERO DE OBJETOS TOTAL INICIAL
numero_objetos_inicialtotal=0;
int j = 0;

for (std::vector<pcl::PointIndices>::const_iterator it = cluster_indices.begin (); it !=
        cluster_indices.end (); ++it)
{
        pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_cluster (new
                pcl::PointCloud<pcl::PointXYZRGB>);

        for (std::vector<int>::const_iterator pit = it->indices.begin (); pit !=
                it->indices.end (); ++pit)
                cloud_cluster->points.push_back (cloud_filtered->points[*pit]); //*

        cloud_cluster->width = cloud_cluster->points.size ();
        cloud_cluster->height = 1;
        cloud_cluster->is_dense = true;

        //Calc point cloud vertexes
        //Primera iteración siempre es menor que los min y mayor que los max
        float x_min = INFINITY;float y_min = INFINITY;float z_min = INFINITY;
        float x_max = -INFINITY;float y_max = -INFINITY;float z_max = -INFINITY;
        float centroid_x=-INFINITY;float centroid_y=-INFINITY;
        float centroid_z=-INFINITY;

        boost::shared_ptr<Eigen::Vector3f> centroid3D (new Eigen::Vector3f());
        *centroid3D = Eigen::Vector3f::Zero();

        for (int i = 0; i < cloud_cluster->points.size(); i++)
        {
                if (cloud_cluster->points[i].x < x_min)
                        x_min = cloud_cluster->points[i].x;
                if (cloud_cluster->points[i].y < y_min)
                        y_min = cloud_cluster->points[i].y;
                if (cloud_cluster->points[i].z < z_min)
                        z_min = cloud_cluster->points[i].z;
                if (cloud_cluster->points[i].x > x_max)
                        x_max = cloud_cluster->points[i].x;
                if (cloud_cluster->points[i].y > y_max)
                        y_max = cloud_cluster->points[i].y;
                if (cloud_cluster->points[i].z > z_max)
                        z_max = cloud_cluster->points[i].z;

                centroid3D->coeffRef(0) += cloud_cluster->points[i].x;
                centroid3D->coeffRef(1) += cloud_cluster->points[i].y;
                centroid3D->coeffRef(2) += cloud_cluster->points[i].z;
        }
        *centroid3D /= (double)cloud_cluster->points.size();
        centroid_x=(*centroid3D)(0); centroid_y=(*centroid3D)(1);
        centroid_z=(*centroid3D)(2);

        //ax + by + cz + d = 0
        z_min=(-D-B*centroid_y-A*centroid_x)/C; //suelo (ecuación del plano)

        if (var == 0) //ES un coche
        {
                float w=x_max-x_min;float h=y_max-y_min;float d=z_max-z_min;
                z_max=z_min+CAR_HEIGHT;

                //Ajustar a caja fija///////
                if (w>CAR_LENGTH)
                {
                        x_max=centroid_x+CAR_LENGTH/2;
                        x_min=centroid_x-CAR_LENGTH/2;
```

```
                    }else if(w<CAR_WIDTH)
                    {
                            x_max=centroid_x+CAR_WIDTH/2;
                            x_min=centroid_x-CAR_WIDTH/2;
                    }
                    if(h>CAR_LENGTH) //Como máximo 3.5x3.5x1.3
                    {
                            y_max=centroid_y+CAR_LENGTH/2;
                            y_min=centroid_y-CAR_LENGTH/2;

                    }else if (h<CAR_WIDTH) //Como mínimo 1.65x1.65x1.3
                    {
                            y_max=centroid_y+CAR_WIDTH/2;
                            y_min=centroid_y-CAR_WIDTH/2;
                    }
                    //FIN Ajustar a caja fija///////

                    w=x_max-x_min; h=y_max-y_min; d=z_max-z_min;

                    car.centroid(0)=centroid_x; car.centroid(1)=centroid_y;
                    car.centroid(2)=centroid_z;
                    car.x_max=x_max; car.x_min=x_min; car.y_max=y_max;
                    car.y_min=y_min; car.z_max=z_max; car.z_min=z_min;
                    car.w=w; car.h=h;car.d=d;
                    car.cloud=cloud_cluster;
                    car_vector.push_back(car); //Introduce el objeto en un vector
                    coche=coche+1;
            }

            if (var == 1) //ES un peatón
            {

                    float w=x_max-x_min; float h=y_max-y_min; float d=z_max-z_min;
                    z_max=z_min+PEDESTRIAN_HEIGHT;

                    //Ajustar a caja fija///////
                    if(h!=PEDESTRIAN_LENGTH)
                    {
                            y_max=centroid_y+PEDESTRIAN_LENGTH/2;
                            y_min=centroid_y-PEDESTRIAN_LENGTH/2;
                    }
                    if (w!=PEDESTRIAN_WIDTH)
                    {
                            x_max=centroid_x+PEDESTRIAN_WIDTH/2;
                            x_min=centroid_x-PEDESTRIAN_WIDTH/2;
                    }

                    //FIN Ajustar a caja fija///////

                    w=x_max-x_min;h=y_max-y_min;d=z_max-z_min;

                    pedestrian.centroid(0)=centroid_x;
                    pedestrian.centroid(1)=centroid_y;
                    pedestrian.centroid(2)=centroid_z;
                    pedestrian.x_max=x_max; pedestrian.x_min=x_min;
                    pedestrian.y_max=y_max; pedestrian.y_min=y_min;
                    pedestrian.z_max=z_max; pedestrian.z_min=z_min;
                    pedestrian.w=w; pedestrian.h=h; pedestrian.d=d;
                    pedestrian.cloud=cloud_cluster;
                    pedestrian_vector.push_back(pedestrian);

                    peaton=peaton+1;
            }
```

```
                        objeto.centroid(0)=centroid_x; objeto.centroid(1)=centroid_y;
                        objeto.centroid(2)=centroid_z;
                        objeto.x_max=x_max; objeto.x_min=x_min; objeto.y_max=y_max;
                        objeto.y_min=y_min; objeto.z_max=z_max; objeto.z_min=z_min;
                        objeto.w=objeto.x_max-objeto.x_min;
                        objeto.h=objeto.y_max-objeto.y_min;
                        objeto.d=objeto.z_max-objeto.z_min;
                        objeto.cloud=cloud_cluster;

                        if(var==0)
                                objeto.tipo="Coche";
                        else if (var==1)
                                objeto.tipo="Peaton";

                        objetos.push_back(objeto); //Introduce struct objeto en vector objetos

                        std::stringstream ss1;
                        ss1 << "cloud_cluster_" << var << j; //Dibuja la caja que contiene los puntos
                        viewer->addCube(x_min,x_max,y_min,y_max,z_min,z_max,r,g,b,ss1.str() );

                        numero_objetos_inicialtotal++;
                        j++;
                }
        }
}
////////////////////////////   FIN   FUNCIONES ////////////////////////////////

void cloud_cb (const sensor_msgs::PointCloud2ConstPtr& msg, const nav_msgs::OdometryConstPtr&
msg1)
{
        // Create a container for the data.
        pcl::PointCloud<pcl::PointXYZRGB>::Ptr vlp_cloud_Ptr (new
                pcl::PointCloud<pcl::PointXYZRGB>);
        pcl::PCLPointCloud2::Ptr Input_Cloud (new pcl::PCLPointCloud2 ());
        pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud (new  pcl::PointCloud<pcl::PointXYZRGB>),
                                                cloud_f (new pcl::PointCloud<pcl::PointXYZRGB>);
        // Do data processing here...
        pcl_conversions::toPCL(*msg,*Input_Cloud);
        pcl::fromPCLPointCloud2(*Input_Cloud, *vlp_cloud_Ptr);

        //Transformación de giro para coincidir con VREP
        Eigen::Affine3f transform = Eigen::Affine3f::Identity();
        transform.translation() << 0.0, 0.0, 0.0; // Define a translation
        // Rotation (VREP Frame))
        float roll, pitch, yaw; roll=M_PI/2;
        transform.rotate (Eigen::AngleAxisf (roll, Eigen::Vector3f::UnitX()));
        pcl::transformPointCloud (*vlp_cloud_Ptr, *vlp_cloud_Ptr, transform);
        ////////////////////////////////////////////////////////////////////////

        pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_filtered (new
                pcl::PointCloud<pcl::PointXYZRGB>);
        pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_plane (new
                pcl::PointCloud<pcl::PointXYZRGB>);

        //VISUALIZAR NUBE SIN FILTRAR
        pcl::visualization::PointCloudColorHandlerRGBField<pcl::PointXYZRGB> handle_raw
                (vlp_cloud_Ptr);
        viewer->addPointCloud<pcl::PointXYZRGB> (vlp_cloud_Ptr,handle_raw, "cloud_raw");

        *cloud_filtered=*vlp_cloud_Ptr;

        // Create the segmentation object for the planar model and set all the parameters
        pcl::SACSegmentation<pcl::PointXYZRGB> seg;
        pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
        pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
```

```
seg.setOptimizeCoefficients (true);
seg.setModelType (pcl::SACMODEL_PLANE);
seg.setMethodType (pcl::SAC_RANSAC);
seg.setMaxIterations (100);
seg.setDistanceThreshold (0.15);

// Segment the largest planar component from the remaining cloud
seg.setInputCloud (cloud_filtered);
seg.segment (*inliers, *coefficients);

if (inliers->indices.size () == 0)
{
	std::cout << "Could not estimate a planar model for the given dataset." << std::endl;
	A=0;B=0;C=1;D=LIDAR_HEIGHT;
}else{// Extract the planar inliers from the input cloud
	pcl::ExtractIndices<pcl::PointXYZRGB> extract;
	extract.setInputCloud (cloud_filtered);
	extract.setIndices (inliers);
	extract.setNegative (false);
	extract.filter (*cloud_plane); // Get the points associated with the planar surface

	// Remove the planar inliers, extract the rest
	extract.setNegative (true);
	extract.filter (*cloud_f);
	*cloud_filtered = *cloud_f;

	A=coefficients->values[0]; B=coefficients->values[1]; C=coefficients->values[2];
	D=coefficients->values[3];
}

//FILTRADO EN COLOR
pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_filtered_red_car (new
	pcl::PointCloud<pcl::PointXYZRGB>);
pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_filtered_pedestrian (new
	pcl::PointCloud<pcl::PointXYZRGB>);

*cloud_filtered_red_car=*cloud_filtered; *cloud_filtered_pedestrian=*cloud_filtered;

segmentation_filter (cloud_filtered_red_car, 255,125,55,0,55,0, 0, 0, 1, 0, 1.3, 20, 350);
segmentation_filter (cloud_filtered_pedestrian,  147,80,165,116,202,146, 0, 1, 0, 1, 0.5, 10, 150);

*cloud_filtered=*cloud_filtered_red_car; *cloud_filtered+=*cloud_filtered_pedestrian;
*cloud_frame=*cloud_filtered; //Evaluacion

//KALMAN PARA LOS OBJETOS
Kalman_State state_kalman_prediction; //1 punto
Array_Kalman_States states_kalman; //10 posiciones
pcl::PointXYZ point_act, point_pre, p1, p2;

for (int n_obj=0; n_obj<objetos.size(); n_obj++) //recorre los objetos que hay en el frame
{
	//Guarda el punto actual como primer punto del array de medidas
	states_kalman.x[0]=objetos[n_obj].centroid(0);
	states_kalman.y[0]=objetos[n_obj].centroid(1);
	states_kalman.z[0]=objetos[n_obj].centroid(2);

	states_kalman.w[0]=objetos[n_obj].w; states_kalman.h[0]=objetos[n_obj].h;
	states_kalman.d[0]=objetos[n_obj].d; point_act.x=objetos[n_obj].centroid(0);
	point_act.y=objetos[n_obj].centroid(1); point_act.z=objetos[n_obj].centroid(2);

	//////////////////

	float dist = 0;
	float minDist=INFINITY;
	int minKalman = -1;
```

```
// Associate available Kalman filters with detected objects (distances between centers)
for (unsigned int n_fil=0; n_fil<kfs.size(); n_fil++) //Recorre el vector de filtros kf
{
        //Identificador del filtro según distancia entre objeto y prediccion
        if (kfsTime[n_fil]>0) // If the kfsTime is higher than 0, the filter is considered
        {
        // Distance between detected object center and the Kalman filters centers
                state_kalman_prediction=getKalmanPrediction(kfs[n_fil]);
                point_pre.x=state_kalman_prediction.x;
                point_pre.y=state_kalman_prediction.y;
                point_pre.z=state_kalman_prediction.z;

                dist = float(sqrt(pow(point_act.x-point_pre.x,2)+pow(point_act.y-
                        point_pre.y,2)));

                if ((dist<minDist)&&(dist<3.5))
                {
                        if (strcmp(objetos[n_obj].tipo,"Coche") == 0)
                        {
                                if ((state_kalman_prediction.w>=1.5) &&
                                        (state_kalman_prediction.h>=1.5))
                                {
                                        minDist = dist;
                                        minKalman = n_fil; //se actualiza
                                }
                        }else{
                                if ((state_kalman_prediction.w<1) &&
                                        (state_kalman_prediction.h<1))
                                {
                                        minDist = dist;
                                        minKalman = n_fil; //se actualiza
                                }
                        }
                }
        }
}

if (minKalman==-1) //Entra cuando no hay ningún filtro asociado
{
        //inicia filtro con valores actuales
        kfs.push_back(initKalman(objetos[n_obj].centroid(0),
        objetos[n_obj].centroid(1), objetos[n_obj].centroid(2),
        objetos[n_obj].w, objetos[n_obj].h, objetos[n_obj].d, 0.999, 0.999,
        0.999, 0.001,0.001, 0.001, 0.01));
        kfs_array.push_back(states_kalman); //Guarda la posicion actual
        kfsTime.push_back(10);//Para cuando pierde el objeto
        minKalman = kfs.size()-1; //Actualiza el identifiador

        //--------------------PRECISION-TRACKER-----------------------------------
        //Añade un nuevo tracker para cada nuevo objeto
        precision_tracking::Tracker tracker(&params);
        tracker.setPrecisionTracker(boost::make_shared
                <precision_tracking::PrecisionTracker>(&params));
        // Reset the tracker for this new track.
        tracker.clear(); //resetea el modelo previo y el actual
        trackers.push_back(tracker); //se guarda en el array de trackers

        //Cada vez que se visualiza el objeto
        precision_tracking::getSensorResolution16(objetos[n_obj].centroid,
        &sensor_horizontal_resolution, &sensor_vertical_resolution);
        trackers[minKalman].addPoints(objetos[n_obj].cloud, frame_time,
                sensor_horizontal_resolution, sensor_vertical_resolution,
                &estimated_velocity);
        precisiontracker.velocity_estimated=estimated_velocity;
        tracker_vector.push_back(precisiontracker);
```

```
}else{  // If the object is associated, Kalman filter is updated
        //Predicción actual (antes de update) ////////////////
        state_kalman_prediction=getKalmanPrediction(kfs[minKalman]);
        point_pre.x=state_kalman_prediction.x;
        point_pre.y=state_kalman_prediction.y+state_kalman_prediction.h/2;
        point_pre.z=state_kalman_prediction.z;

        float state_kalman_prediction_x_min=state_kalman_prediction.x-
                                                state_kalman_prediction.w/2;
        float state_kalman_prediction_x_max=state_kalman_prediction.x+
                                                state_kalman_prediction.w/2;
        float state_kalman_prediction_y_min=state_kalman_prediction.y-
                                                state_kalman_prediction.h/2;
        float state_kalman_prediction_y_max=state_kalman_prediction.y+
                                                state_kalman_prediction.h/2;
        float state_kalman_prediction_z_min=state_kalman_prediction.z-
                                                state_kalman_prediction.d/2;
        float state_kalman_prediction_z_max=state_kalman_prediction.z+
                                                state_kalman_prediction.d/2;

        viewer->addCube(state_kalman_prediction_x_min,
        state_kalman_prediction_x_max, state_kalman_prediction_y_min,
        state_kalman_prediction_y_max, state_kalman_prediction_z_min,
        state_kalman_prediction_z_max, 1,1,1,ss4.str());

        //Actualiza el estado predicho con la medida ///////////////
        updateKalman(kfs[minKalman], objetos[n_obj].centroid(0),
                objetos[n_obj].centroid(1), objetos[n_obj].centroid(2),
                objetos[n_obj].w, objetos[n_obj].h, objetos[n_obj].d, true);
        kfsTime[minKalman]=10; //REcupera

        updateKalman_shift(minKalman,objetos[n_obj].centroid(0),
                objetos[n_obj].centroid(1), objetos[n_obj].centroid(2),
                objetos[n_obj].w, objetos[n_obj].h, objetos[n_obj].d, frame_time);
        //-------------------PRECISION-TRACKER---------------------------------
        //Para cada vez que aparece el objeto en escena
        precision_tracking::getSensorResolution16(objetos[n_obj].centroid,
                &sensor_horizontal_resolution, &sensor_vertical_resolution);
        trackers[minKalman].addPoints(objetos[n_obj].cloud, frame_time,
                &sensor_horizontal_resolution, sensor_vertical_resolution,
                &estimated_velocity);
        tracker_vector[minKalman].velocity_estimated=estimated_velocity;
}

for (unsigned int i=0; i<kfs.size(); i++)
{
        if((kfsTime[i]>0) && (i!=minKalman))
        {
                Kalman_State kalman_prediction=getKalmanPrediction(kfs[i]);
                updateKalman(kfs[i], kalman_prediction.x, kalman_prediction.y,
                                kalman_prediction.z, kalman_prediction.w,
                                kalman_prediction.h, kalman_prediction.d, false);
        }
}

//-------------------------------MOSTRAR POR PANTALLA-------------------------------
velocidad_real=float(3.6*sqrt(pow(gt_vel(0),2)+pow(gt_vel(1),2)+pow(gt_vel(2),2)));
velocidad_kalman= float(3.6*sqrt(pow(kf_vel(0),2)+pow(kf_vel(1),2)+
                pow(kf_vel(2),2)));
velocidad_precision=
        float(3.6*sqrt(pow(tracker_vector[minKalman].velocity_estimated(0),2)
        +pow(tracker_vector[minKalman].velocity_estimated(1),2)
        +pow(tracker_vector[minKalman].velocity_estimated(2),2)));
```

```
                kf_rvel(0)=float(gt_vel(0)+kf_vel(0)); kf_rvel(1)=float(gt_vel(1)+kf_vel(1));
                kf_rvel(2)=0;
                pt_rvel(0)=float(gt_vel(0)+tracker_vector[minKalman].velocity_estimated(0));
                pt_rvel(1)=float(gt_vel(1)+tracker_vector[minKalman].velocity_estimated(1));
                pt_rvel(2)=0;

                float vel_rel_kf=float(3.6*sqrt(pow(kf_rvel(0),2)+pow(kf_rvel(1),2)+
                                pow(kf_rvel(2),2)));
                float vel_rel_pt=float(3.6*sqrt(pow(pt_rvel(0),2)+pow(pt_rvel(1),2)+
                                pow(pt_rvel(2),2)));

////////////////////////////////////////////////////////////////////////////
                pcl::PointXYZ p_future_kf, p_obj;
                p_obj.x=point_act.x; p_obj.y=point_act.y; p_obj.z=point_act.z;
                p_future_kf.x=point_act.x+kf_vel(0)*0.5; //At 0.5s
                p_future_kf.y=point_act.y+kf_vel(1)*0.5;
                p_future_kf.z=point_act.z;
                std::stringstream ss_pre;
                ss_pre << "prediction_precision"<<n_obj<<minKalman;

                pcl::PointXYZ p_future_preci;
                p_future_preci.x=point_act.x+
                                tracker_vector[minKalman].velocity_estimated(0)*0.5;
                p_future_preci.y=point_act.y+
                                tracker_vector[minKalman].velocity_estimated(1)*0.5;
                p_future_preci.z=point_act.z;
                viewer->addLine(point_act,p_future_preci,0,0,1,ss_pre.str());

                //pintar la trayectoria del filtro
                for (unsigned int i=0; i<kfs.size(); i++)
                {
                        if(kfsTime[i]>0)
                        {
                                for(int j=0;j<9;j++)
                                {
                                        p1.x=kfs_array[i].x[j]; p2.x=kfs_array[i].x[j+1];
                                        p1.y=kfs_array[i].y[j]; p2.y=kfs_array[i].y[j+1];
                                        p1.z=kfs_array[i].z[j]-kfs_array[i].h[j]/2;
                                        p2.z=kfs_array[i].z[j+1]-kfs_array[i].h[j+1]/2; //suelo

                                        if(p2.x!=0 &&  p2.y!=0 && p2.z!=0)
                                                viewer->addLine(p1,p2,1,1,1,ss6.str()); //white
                                }
                        }
                }

        }//fin for objetos

        if (objetos.size()==0) //Actualizar todos los filtros con su predicción
        {
                for (unsigned int i=0; i<kfs.size(); i++)
                {
                        if(kfsTime[i]>0)
                        {
                                Kalman_State kalman_prediction=getKalmanPrediction(kfs[i]);
                                updateKalman(kfs[i], kalman_prediction.x, kalman_prediction.y,
                                                kalman_prediction.z,
                                kalman_prediction.w, kalman_prediction.h, kalman_prediction.d,
                                                false); //Actualizar con su predicion
                        }
                }
        }
```

```
	// Decrease the time of life of Kalman filters
	for (unsigned int i=0; i<kfsTime.size(); i++)
	{
		kfsTime[i]--;
	}
		//Eliminar el filtro de kalman si no se trabaja con él
		int offset=0; //Para que no se salte iteraciones cuando borra una
		for (unsigned int n_fil=0; n_fil<kfs.size(); n_fil++)
		{
			if (kfsTime[n_fil]<=0) // If the kfsTime is lower 0, the filter and counter are erased
			{
				kfsTime.erase(kfsTime.begin()+n_fil-offset);
				kfs.erase(kfs.begin()+n_fil-offset);
				kfs_array.erase(kfs_array.begin()+n_fil-offset);
				tracker_vector.erase(tracker_vector.begin()+n_fil-offset);
				trackers.erase(trackers.begin()+n_fil-offset);
				offset++;
				for (unsigned int i=0; i<kfs.size(); i++)
				{
					if(i>=n_fil)
				}
				file.close();
			}
		}

		////////// PUBLICAR TOPIC  PUNTOS OBSTÁCULOS ////////////////////////
		visualization_msgs::Marker points;
		geometry_msgs::Point p;
		points.header.frame_id = "/base_link"; points.header.stamp = ros::Time::now();
		points.ns = "Marcador"; points.action = visualization_msgs::Marker::ADD;
		points.pose.orientation.w = 1.0; points.id = 0;
		points.type = visualization_msgs::Marker::SPHERE_LIST; //POINTS
		// POINTS markers use x and y scale for width/height respectively
		points.scale.x = 1; points.scale.y = 1; points.scale.z = 1;
		// Points are yellow
		points.color.r = 1.0f; points.color.g = 1.0f; points.color.a = 1.0f;

		points.points.clear();

		for (int n_obj=0; n_obj<objetos.size(); n_obj++)
		{
			p.x=objetos[n_obj].centroid(0); p.y=objetos[n_obj].centroid(1); p.z=0;
			points.points.push_back(p);
		}
		obstacle_pub_.publish(points);
}

////////////////////////////////////////////////////////////////////////////////
void callback(const sensor_msgs::PointCloud2::ConstPtr& msg, const nav_msgs::Odometry::ConstPtr&
msg1)
{
		clock_t startTime = clock(); //Start timer
		double secondsPassed;
		frame_time=msg->header.stamp.toSec();
		//Ground-truth
		gt_vel(0)=msg1->twist.twist.linear.x; gt_vel(1)=msg1->twist.twist.linear.y;
		gt_vel(2)=msg1->twist.twist.linear.z;
		//Borra en cada iteración
		viewer->removeAllPointClouds (); viewer->removeAllShapes();
		objetos.clear();
		cloud_cb(msg, msg1);

		//TIEMPO DE EJECUCIÓN
		secondsPassed = ( std::clock() - startTime ) / (double) CLOCKS_PER_SEC;
}
```

```
// CALLBACK DEL TIMER
void spin(const ros::TimerEvent& e)
{
	viewer->spinOnce (5); //5ms
}

// PROGRAMA PRINCIPAL
int main(int argc, char **argv)
{
	// INICIALIZAR ROS
	ros::init(argc, argv, "main_node");

	// MANEJADOR DE NODOS DE ROS
	ros::NodeHandle nh;

	obstacle_pub_ = nh.advertise<visualization_msgs::Marker>("/Obstacle_marker", 1, true);

	//MESSAGE_FILTERS
	//Suscripción a cada topic
	message_filters::Subscriber<sensor_msgs::PointCloud2> lidar_sub_(nh,
					"/velodyne_coloured_points", 1);//Nube coloreada por but_velodyne
	message_filters::Subscriber<nav_msgs::Odometry> odom_sub_(nh, "/odom", 1); //Ground-truth

	//Sincronización de datos recibidos y llamada a callback caa vez que se sincronizan
	typedef sync_policies::ApproximateTime<sensor_msgs::PointCloud2,
					nav_msgs::Odometry> MySyncPolicy;

	// ExactTime takes a queue size as its constructor argument, hence MySyncPolicy(10)
	Synchronizer<MySyncPolicy> sync(MySyncPolicy(10), lidar_sub_, odom_sub_);
	sync.registerCallback(boost::bind(&callback, _1, _2)); //2 topics

	viewer->setBackgroundColor (0, 0, 0);
	viewer->addCoordinateSystem (1.0);
	viewer->initCameraParameters ();
	viewer->setCameraPosition (-3.73, -0.1, 0.59, -0.11, -0.04, 1); //POV

	// TIMER
	timer_ = nh.createTimer(ros::Duration(0.05), &spin);

	// LANZAR EL NODO
	ros::spin();
}
```

PLIEGO DE CONDICIONES

En este apartado se incluirán las principales herramientas hardware y software empleadas para el desarrollo del trabajo.

PC.1. Hardware

- **Portátil *ASUS S551LN*:**
  - Procesador: *Intel® Core™ i7*
  - Memoria RAM: *8GB*
  - Tarjeta gráfica: *NVIDIA® GeForce 840M*
  - Disco Duro: *SSD 256 GB + HDD 1 TB*

- **Vehículo Eléctrico Open Source *Tabby Evo*:**
  - Velocidad máxima: *100 km/h*
  - Autonomía: *80 km*
  - Potencia: *19 kW*
  - Peso: *380 kg*

- ***Velodyne LiDAR Puck* (VLP-16)*:***
  - Canales: *16*
  - Rango máximo de medida: *100m*
  - Máximo número de puntos/segundo: *600.000*
  - Precisión: *3cm*

- **Cámara estéreo *Bumblebee® XB3***:
  - Sensor: *3 CCD Sony ICX445 1/3”*
  - Tamaño de imagen máximo: *1280x960*
  - Distancia focal: *3.8mm (66° HFOV) o 6mm (43° HFOV)*
  - Resolución del convertidor A/D: *12 bits*

- **GPS *HIPer Pro de Topcon***:
  - Canales de búsqueda: *20 GPS L1+L2 (GD), GPS L1 + GLONASS (GG)*
    20 GPS L1+L2+GLONASS (GGD)
  - Comunicación: *Bluetooth® versión 1.1 comp.*
  - Puertos I/O: *2x serie (RS232)*
  - Velocidad de salida: *20Hz*

PC.2. Software

- Ubuntu 14.04.5 LTS (Trusty Tahr).
- ROS Indigo Igloo.
- V-REP PRO EDU V3.4.0.
- Point Cloud Library V1.7.
- Paquetes de ROS.
- Microsoft Office 365 ProPlus.

PRESUPUESTO

En este capítulo se proporciona información detallada sobre los costes teóricos del desarrollo del proyecto, incluyendo los costes de materiales y las tasas profesionales.

PR.1. Costes Materiales

En la siguiente tabla se detallan los costes de las herramientas hardware y software empleadas para el desarrollo del proyecto sin incluir el IVA.

Concepto		**Unidades**	**Coste unitario [€]**	**Coste Total [€]**
Hardware	Portátil ASUS	1	999,00	999,00
	Vehículo Open Source Tabby Evo	1	20.250,00	20.250,00
	Velodyne LiDAR Puck (VLP-16)	1	7.560,00	7.560,00
	Cámara estéreo Bumblebee® XB3	1	3.000,00	3.000,00
	GPS HIPer Pro de Topcon	1	2.660,00	2.660,00
Software	Ubuntu 14.04.5 LTS (Trusty Tahr)	1	0,00[9]	0,00
	ROS Indigo Igloo	1	0,00	0,00
	V-REP PRO EDU V3.4.0	1	0,00	0,00
	Point Cloud Library V1.7.	1	0,00	0,00
	Paquetes de ROS	1	0,00	0,00
	Microsoft Office 365 ProPlus	1	0,00	0,00
TOTAL [€]				**34.469,00**

TABLA PR.1-1: Costes materiales (hardware y software) sin IVA**.**

[9] Los precios unitarios son gratuitos debido a versiones de estudiantes o a software libre.

PR.2. Tasas Profesionales

En la siguiente tabla se detallan las tasas profesionales derivadas de la ejecución del presente trabajo sin incluir el IVA.

Concepto	**Horas**	**Precio unitario [€]**	**Precio Total [€]**
Honorarios de ejecución y dirección de ingeniería	460	12,50	5.750,00
Honorarios de redacción del libro	85	8,00	680,00
TOTAL [€]			**6.430,00**

TABLA PR.2-1: Tasas profesionales de ejecución del proyecto (sin IVA).

PR.3. Costes Totales

Los costes totales del proyecto han sido obtenidos sumando los costes materiales y profesionales aplicando el IVA.

Concepto	**Costes Totales [€]**
Costes materiales	**34.469,00**
Costes profesionales	**6.430,00**
Subtotal	**40.899,00**
IVA (21%)	**8.588,79**
TOTAL [€]	**49.487,79**

TABLA PR.3-1: Costes totales con IVA.

MANUAL DE USUARIO

En esta sección se explicará la guía de instalación que debe seguirse, así como el proceso de ejecución de programas que debe llevarse a cabo para poder probar el sistema descrito en este trabajo.

MU.1. Guía de instalación

Como ya se ha comentado a lo largo del trabajo, el sistema expuesto se ha implementado en la plataforma ROS, haciendo uso de diferentes paquetes que se van a detallar a continuación. Para que la instalación sea efectiva, y garantizar el correcto funcionamiento, se parte de un ordenador sin ningún tipo de sistema operativo, ni programa preinstalado. A continuación, se detallan los pasos que se deben seguir:

PASO 0: INSTALACIÓN DEL SISTEMA OPERATIVO

El Sistema Operativo que va a emplearse es Ubuntu 14.04.5 LTS (Trusty) por su compatibilidad con los diferentes paquetes. Este puede descargarse de manera gratuita desde la página oficial de Ubuntu como una imagen ISO:

Link de descarga Ubuntu 14.04.5 LTS (Trusty): http://releases.ubuntu.com/14.04/

En el presente trabajo se ha descargado para una arquitectura x64 y se ha hecho la instalación mediante USB booteable, realizando un Dual-Boot con Windows. Sin embargo, este método de instalación no es único ni necesario y se deja a libre elección del usuario.

PASO 1: INSTALACIÓN ROS INDIGO

La versión de ROS empleada es ROS Indigo que es compatible con Ubuntu Trusty. La instalación se realiza desde el Terminal de Ubuntu, y se siguen los pasos explicados en el siguiente enlace:

Pasos para la instalación de ROS Indigo: http://releases.ubuntu.com/14.04/

1. **Configurar los repositorios de Ubuntu**: Deben habilitarse las opciones de "restricted", "universe," y "multiverse", como muestra la imagen a continuación:

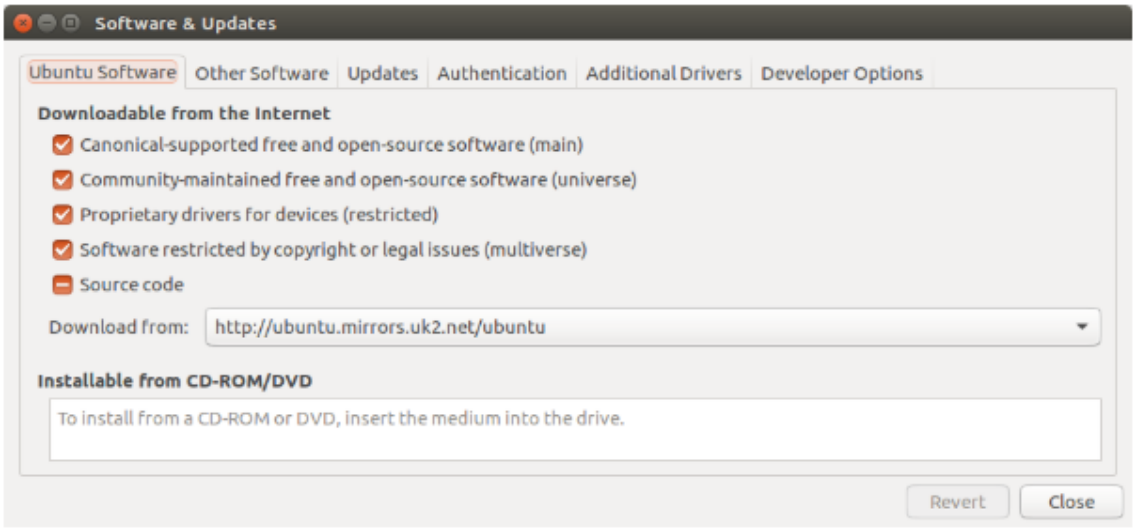

FIGURA MU.1-1: Habilitar los repositorios de Ubuntu.

Se puede acceder a esta interfaz dentro de "Ubuntu Software Center", desde el menú "Edit" clickando en "Software Sources".

2. **Configurar el ordenador para aceptar software desde packages.ros.org:**

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc)
main" > /etc/apt/sources.list.d/ros-latest.list'
```

3. **Configurar las *keys*:**

```
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key
421C365BD9FF1F717815A3895523BAEEB01FA116
```

4. **Instalación de ROS:**

Primero para asegurar que el índice de Debian se encuentra actualizado se ejecuta el siguiente comando:

```
sudo apt-get update
```

Se elige la opción de Instalación Completa que incluye rqt, rviz, librerías genéricas de robótica y simuladores 2D/3D:

```
$ sudo apt-get install ros-indigo-desktop-full
```

5. **Habilitar las dependencias:** necesario para ejecutar algunos componentes de ROS y para instalar dependencias que se quieran compilar desde archivos fuente.

```
$ sudo rosdep init
$ rosdep update
```

6. **Configurar el entorno:** cada vez que se abre un nuevo terminal conviene que las variables se añadan automáticamente al *bash*:

```
$ echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

7. **Descargar herramientas de ROS:**

```
$ sudo apt-get install python-rosinstall ros-indigo-catkin
```

PASO 3: INSTALACIONES PREVIAS NECESARIAS

Para evitar problemas a la hora de compilar los paquetes de ROS es necesario realizar las siguientes instalaciones:

```
$ sudo apt-get update
$ sudo apt-get install build-essential checkinstall cmake libpcap-dev
libpcap0.8-dev libeigen3-dev ros-indigo-navigation ros-indigo-geodesy
```

PASO 4: INSTALACIÓN DE VREP

El entorno de simulación empleado será VREP 3.4.0 en su versión PRO-EDU, que se trata de una versión educacional gratuita no limitada. Para su instalación se descargará escogiendo la opción para Linux x64, desde el siguiente enlace, y se ejecutará el instalador:

Link de descarga VREP 3.4.0 PRO-EDU: http://www.coppeliarobotics.com/downloads.html

Para poder comunicar VREP con ROS será necesario copiar en la carpeta de instalación los siguientes archivos, ya modificados, que se adjuntarán con el proyecto:

libv_repExtROSInterface.so
libv_repExtSmarteldcar.so
libv_repExtVelodyne.so

PASO 4: INSTALACIÓN DE PCL

Para poder trabajar con las nubes de puntos se instalará *Point Cloud Library (PCL)* en su versión 1.7. Es posible su instalación en Ubuntu a través de PPA:

```
$ sudo add-apt-repository ppa:v-launchpad-jochen-sprickerhof-de/pcl
$ sudo apt-get update
$ sudo apt-get install libpcl-all
```

PASO 5: INSTALACIÓN DE OPENCV

Para poder trabajar con la imagen obtenida por la cámara es necesario instalar la biblioteca de funciones OpenCV en su versión 2.4.9. (necesaria para la instalación del paquete *but_velodyne*). Estas librerías se pueden descargar de forma gratuita desde la página oficial:

Link de descarga OpenCV-2.4.9: https://opencv.org/releases.html

El proceso de instalación seguido es el que se muestra en el siguiente enlace:

Pasos para la instalación de OpenCV-2.4.9: https://www.samontab.com/web/2014/06/installing-opencv-2-4-9-in-ubuntu-14-04-lts/

1. **Instalar dependencias**:

```
$ sudo apt-get update

$ sudo apt-get install libgtk2.0-dev libjpeg-dev libtiff4-dev libjaspe
r-dev libopenexr-dev cmake python-dev python-numpy python-tk libtbb-de
v yasm libfaac-dev libopencore-amrnb-dev libopencore-amrwb-dev libtheo
ra-dev libvorbis-dev libxvidcore-dev libx264-dev libqt4-dev libqt4-ope
ngl-dev sphinx-common texlive-latex-extra libv4l-dev libdc1394-22-dev
libavcodec-dev libavformat-dev libswscale-dev default-jdk ant libvtk5-
qt4-dev
```

2. **Generar el Makefile:**

```
$ wget http://sourceforge.net/projects/opencvlibrary/files/opencv-unix
/2.4.9/opencv-2.4.9.zip

$ unzip opencv-2.4.9.zip

$ cd opencv-2.4.9

$ mkdir build

$ cd build

$ cmake -D WITH_TBB=ON -D BUILD_NEW_PYTHON_SUPPORT=ON -D WITH_V4L=ON -
D INSTALL_C_EXAMPLES=ON -D INSTALL_PYTHON_EXAMPLES=ON -D BUILD_EXAMPLE
S=ON -D WITH_QT=ON -D WITH_OPENGL=ON -D WITH_VTK=ON ..
```

3. **Compilar e instalar:**

```
$ make

$ sudo make install
```

4. **Configurar OpenCv:**

```
$ sudo gedit /etc/ld.so.conf.d/opencv.conf
```

Añadir la siguiente línea al final del archivo y guardarlo:

```
/usr/local/lib
```

Ejecutar las siguientes líneas de código:

```
$ sudo ldconfig
$ sudo gedit /etc/bash.bashrc
```

Añadir las siguientes líneas al final del archivo y guardarlo:

```
PKG_CONFIG_PATH=$PKG_CONFIG_PATH:/usr/local/lib/pkgconfig
export PKG_CONFIG_PATH
```

5. **Finalizar:** Reiniciar el ordenador.

PASO 6: CONFIGURAR ENTORNO DE TRABAJO EN ROS

Antes de proceder a la instalación de los paquetes de ROS que se van a emplear, es necesario configurar el entorno de trabajo en ROS:

1. **Creación de un espacio de trabajo (*workspace*):**

Pasos para la creación de un *workspace* en ROS:

http://wiki.ros.org/catkin/Tutorials/create_a_workspace

En primer lugar, se ejecutan las siguientes líneas de código:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
$ source devel/setup.bash
```

Esto creará 3 carpetas (src, devel y *build*) y un archivo *CMakeList.txt* que estará enlazado a la carpeta *src*. Para garantizar que el *workspace* está bien configurado, es necesario asegurarse que la variable de entorno *ROS_PACKAGE_PATH* incluye el directorio actual.

```
$ echo $ROS_PACKAGE_PATH
/home/youruser/catkin_ws/src:/opt/ros/indigo/share
```

2. Creación de un paquete de ROS:

Como ya se ha explicado en capítulos anteriores, los programas de ROS se agrupan en paquetes. Como para implementar el sistema descrito se emplearán nodos, launch y ficheros de configuración, será necesario crear un paquete que los contenga. Para la creación de un paquete se seguirán los pasos descritos en la wiki de ROS:

Pasos para la creación de un paquete en ROS: http://wiki.ros.org/ROS/Tutorials/CreatingPackage

Los paquetes se crean dentro del directorio "/src" del workspace que se ha creado en el anterior apartado, ejecutando por terminal las siguientes instrucciones:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg main_pkg std_msgs rospy roscpp tf
```

Los argumentos que se añaden después del nombre (main_pkg) son paquetes y librerías de las que depende (en este caso dependerá de las librerías estándar, librerías de C++ y Python y el paquete de transformadas).

Un paquete por defecto va a incluir los siguientes archivos:

- **src/**: directorio donde se encuentran los archivos fuente del paquete. En este directorio se creará el nodo principal de ejecución del programa.
- **include/**: directorio donde se encuentran los ficheros cabecera de los que depende el paquete.
- **msg/**: directorio donde se encuentran los tipos de mensaje definidos por el paquete.
- **launch/**: directorio donde se encuentran los archivos *launch* que permiten lanzar tanto nodos como ficheros de configuración.

Además, incluirá un *CMakeList.txt* necesario para compilar mediante *CMake* y un *package.xml* que describe el paquete y sus dependencias.

PASO 7: DESCARGAR LOS PAQUETES

Una vez creado un espacio de trabajo se procede a la instalación de los paquetes que se emplearán, que deben descargarse y descomprimirse en la carpeta "/src".

- **Paquete but_velodyne para fusión sensorial:**

Primero debe descargarse el paquete but_velodyne_lib de la siguiente dirección:

Link de descarga paquete but_velodyne_lib: https://github.com/robofit/but_velodyne_lib

Deben ejecutarse las siguientes líneas en la terminal:

```
$ cd but_velodyne_lib
$ mkdir bin
$ cd bin
$ cmake ..
$ make
$ sudo make install
```

A continuación, se descargará el paquete *but_velodyne* del siguiente enlace:

Link de descarga paquete but_velodyne: https://github.com/robofit/but_velodyne

Y únicamente deberá compilarse el *workspace* ejecutando:

```
$ cd ~/catkin_ws
$ catkin_make
```

- **Paquete precisión-tracking para seguimiento de velocidad:**

Deberá descargarse el paquete de la siguiente dirección:

Link de descarga paquete precisión-tracking: https://github.com/davheld/precision-tracking

En el archivo package.xml del paquete deberán añadirse las siguientes líneas de código:

```
<build_depend>cmake_modules</build_depend>
<run_depend>cmake_modules</run_depend>
```

Dentro de *CMakeList.txt* del paquete se añadirá la siguiente línea:

```
find_package(cmake_modules REQUIRED)
```

A continuación, deberán ejecutarse las siguientes líneas en la terminal:

```
$ cd precision-tracking-master
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
```

Por último, debe compilarse el workspace:

```
$ cd ~/catkin_ws
$ catkin_make
```

- **Paquete velo2cam para fusión sensorial:**

Primero debe descargarse el paquete velo2cam de la siguiente dirección:

Link de descarga paquete velo2cam_calibration: https://github.com/beltransen/velo2cam_calibration/tree/master

Y únicamente deberá compilarse el *workspace* ejecutando:

```
$ cd ~/catkin_ws
$ catkin_make
```

MU.2. Guía de ejecución

Una vez seguido el proceso de instalación se puede realizar la ejecución del sistema implementado. Se copiarán los archivos *launch* dentro de la carpeta "/launch" y el nodo principal implementado (.cpp) dentro de la carpeta "/src" del paquete creado.

Para simplificar el lanzamiento de la aplicación, se ha desarrollado un scrip (extensión sh) que lanza secuencialmente cada una de las instrucciones necesarias. Se guardará dentro del directorio "/src" del paquete creado y se ejecutará del siguiente modo:

```
$ cd ~/catkin_ws/src ./lanzador
```

A pesar de que comparten cosas en común, debe diferenciarse el proceso de ejecución en simulación del real como se muestra a continuación.

MU.2.1. Simulación

El script de ejecución de simulación lanzará las instrucciones que se muestran a continuación:

1. Cargar rviz y los ficheros de configuración:

```
$ roslaunch catkin_ws navigation.launch
```

Debe tenerse en cuenta que al lanzar primero el *launch*, se ejecuta automáticamente el *roscore* y no es necesario ejecutarlo en otra terminal.

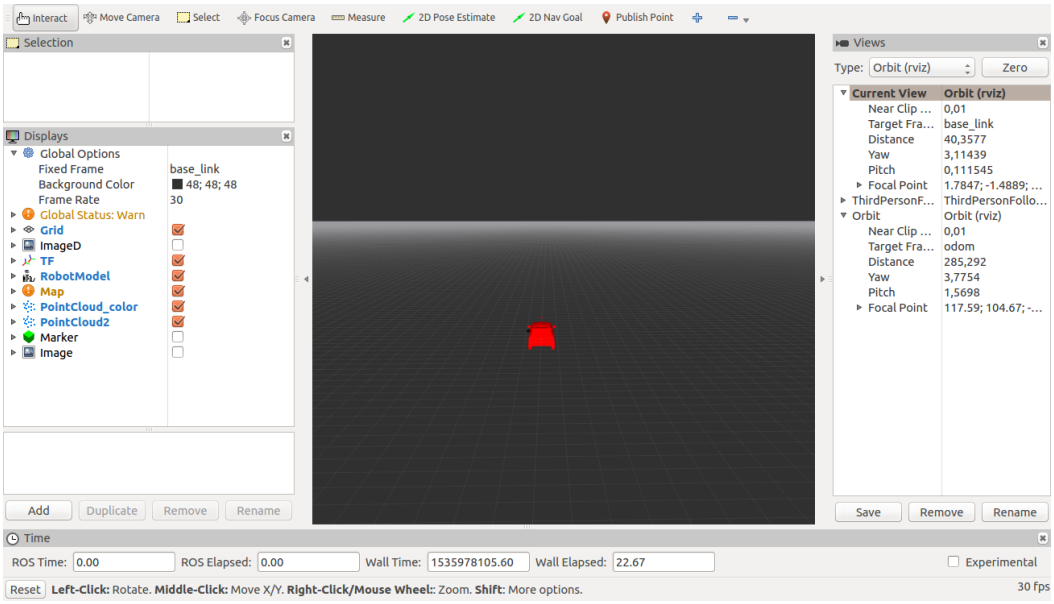

FIGURA MU.2-1: Pantalla incial rviz.

2. Cargar VREP:

```
$ cd ~/VREP ./vrep.sh
```

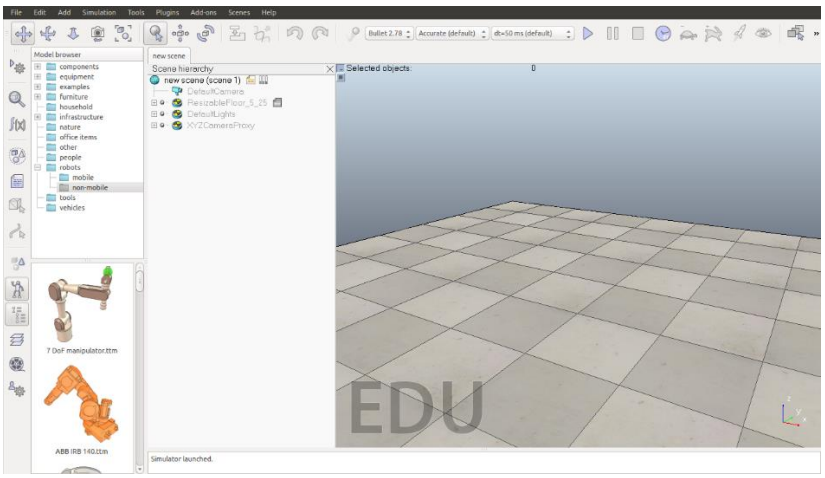

FIGURA MU.2-2: Pantalla inicial VREP.

3. Ejecutar el paquete but_velodyne:

```
$ roslaunch but_calibration_camera_velodyne coloring_vlp16.launch
```

```
SUMMARY
========

PARAMETERS
 * /but_calibration_camera_velodyne/6DoF: [0.0, 0.098, 0.0,...
 * /but_calibration_camera_velodyne/camera_frame_topic: /camera/left/imag...
 * /but_calibration_camera_velodyne/camera_info_topic: /camera/left/came...
 * /but_calibration_camera_velodyne/marker/circles_distance: 0.23
 * /but_calibration_camera_velodyne/marker/circles_radius: 0.0825
 * /but_calibration_camera_velodyne/velodyne_color_topic: /velodyne_coloure...
 * /but_calibration_camera_velodyne/velodyne_topic: /velodyne_points_...
 * /rosdistro: indigo
 * /rosversion: 1.11.21

NODES
  /
    coloring_vlp16 (but_calibration_camera_velodyne/coloring_vlp16)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[coloring_vlp16-1]: started with pid [10772]
```

FIGURA MU.2-3: Terminal inicial but_velodyne.

4. Ejecutar el nodo principal:

```
$ rosrun catkin_ws tracking_object_vlp16.cpp
```

El nodo se mantendrá a la espera sin ejecutarse hasta que se cargue una escena en VREP y se dé al *play*. Para cargar una escena, dentro de la interfaz de VREP se realiza la siguiente secuencia:

```
"File" -> "Open scene…" -> Buscar la escena -> "Aceptar"
```

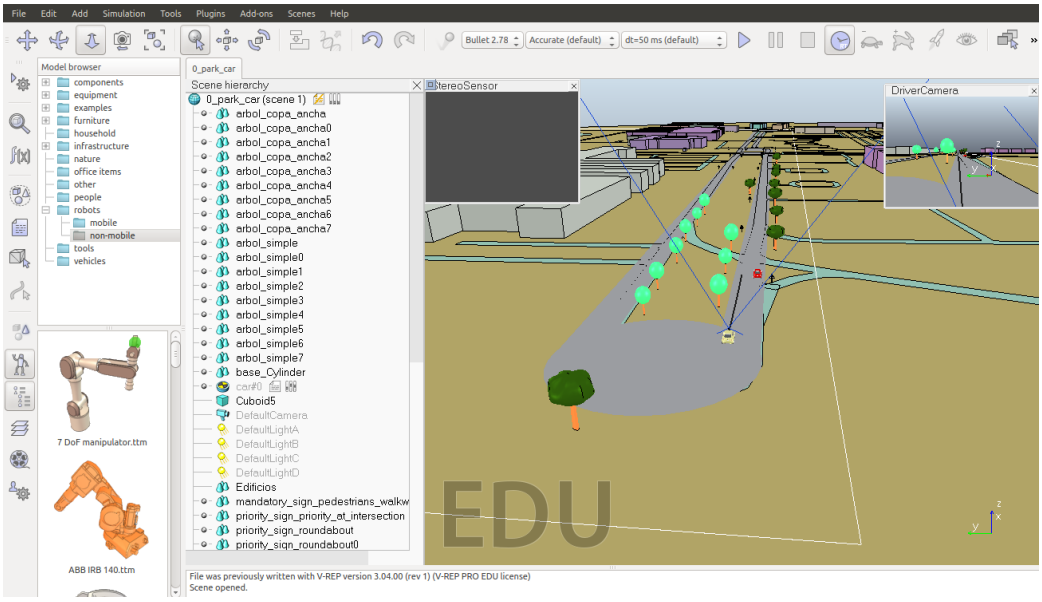

FIGURA MU.2-4: Ejemplo de escena cargada en VREP.

Con la escena ya cargada existe la posibilidad de modificar los elementos en ella:

- Pueden añadirse y eliminarse elementos copiando y pegando sobre la escena.

- Es posible mover la posición de los objetos clickando sobre ellos y pulsando sobre los siguientes botones de la barra de menú:

FIGURA MU.2-5: Botones para mover objetos en VREP.

- Pueden modificarse las trayectorias descritas tanto por el propio vehículo, como por los objetos del entorno, clickando sobre los objetos y pulsando sobre el siguiente botón de la barra lateral de menú:

FIGURA MU.2-6: Botón para modificar trayectoria de objetos en VREP.

  Tras pulsar este botón se abrirá un nuevo menú que permitirá añadir/quitar puntos y modificarlos para definir una trayectoria.

- Una vez que la escena se encuentra a gusto del usuario el siguiente paso es pulsar sobre el *play*:

FIGURA MU.2-7: Botones para iniciar/pausar la escena en VREP.

MU.2.2. Real

El script de ejecución real lanzará las instrucciones que se muestran a continuación:

1. **Cargar rviz y los ficheros de configuración:**

```
$ roslaunch catkin_ws navigation.launch
```

2. Ejecutar el paquete velo2cam:

```
$ roslaunch velo2cam_calibration pcl_coloring_real.launch
```

```
SUMMARY
========

PARAMETERS
 * /pcl_coloring_real/source_frame: velodyne
 * /pcl_coloring_real/target_frame: stereo_camera_LR
 * /rosdistro: indigo
 * /rosversion: 1.11.21

NODES
  /
    pcl_coloring_real (velo2cam_calibration/pcl_coloring_real)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[pcl_coloring_real-1]: started with pid [2790]
[ INFO] [1537382170.482876813]:

Colouring VELODYNE CLOUD!!
FRAME ID velodyne
```

FIGURA MU.2-8: Terminal inicial velo2cam.

5) Ejecutar el nodo principal:

```
$ rosrun catkin_ws tracking_object_real.cpp
```

El nodo se mantendrá a la espera sin ejecutarse hasta que se ejecute el bag con los datos reales. Para ejecutarlo se escribirá en otra terminal la siguiente instrucción:

```
$ rosbag play bag -loop --clock
```

```
Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
[RUNNING]  Bag Time: 1522840999.652306   Duration: 28.934056 / 41.599103
```

FIGURA 2-9: Terminal de ejecución del bag.

MU.2.3. Información por pantalla

Tanto como si se ejecuta la simulación, como si se realiza el caso real, podremos ver la siguiente información:

- **Rviz**: muestra la visualización 3D de la escena. Permite decidir qué topics de los que se están publicando quieren visualizarse y modificar sus características.

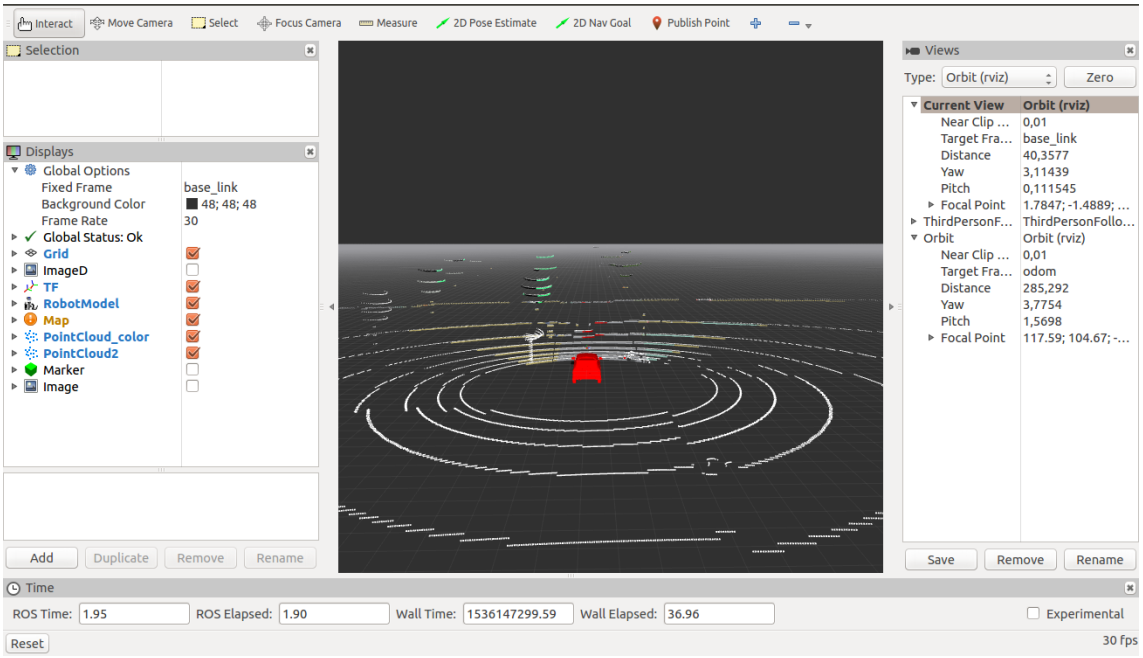

FIGURA MU.2-10: Pantalla de la herramienta rviz en ejecución.

- **Visualizador 3D de PCL**: aporta visualmente la información de la detección, con un cubo alrededor de los objetos detectados; y del seguimiento con la información del filtro que representa cada objeto y la velocidad obtenida mediante precisión-tracking. También dibuja la trayectoria descrita por cada objeto y la dirección de avance.

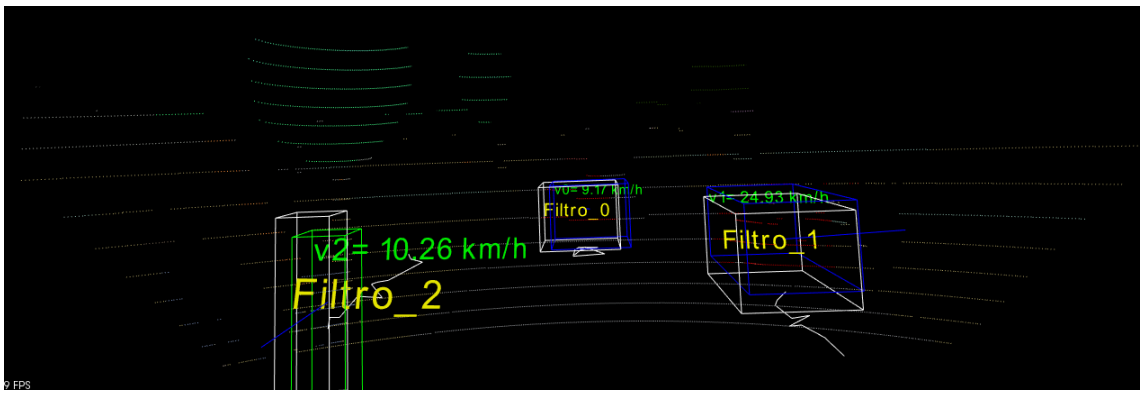

FIGURA MU.2-11: Visualizador 3D de PCL.

- **Terminal de ejecución del nodo principal**: muestra de forma numérica, por cada iteración, la información del número de puntos, antes y después de realizar el filtrado, así como el número de filtros de Kalman existentes y el que se encuentra en ese momento en ejecución, junto con las medidas de velocidad.

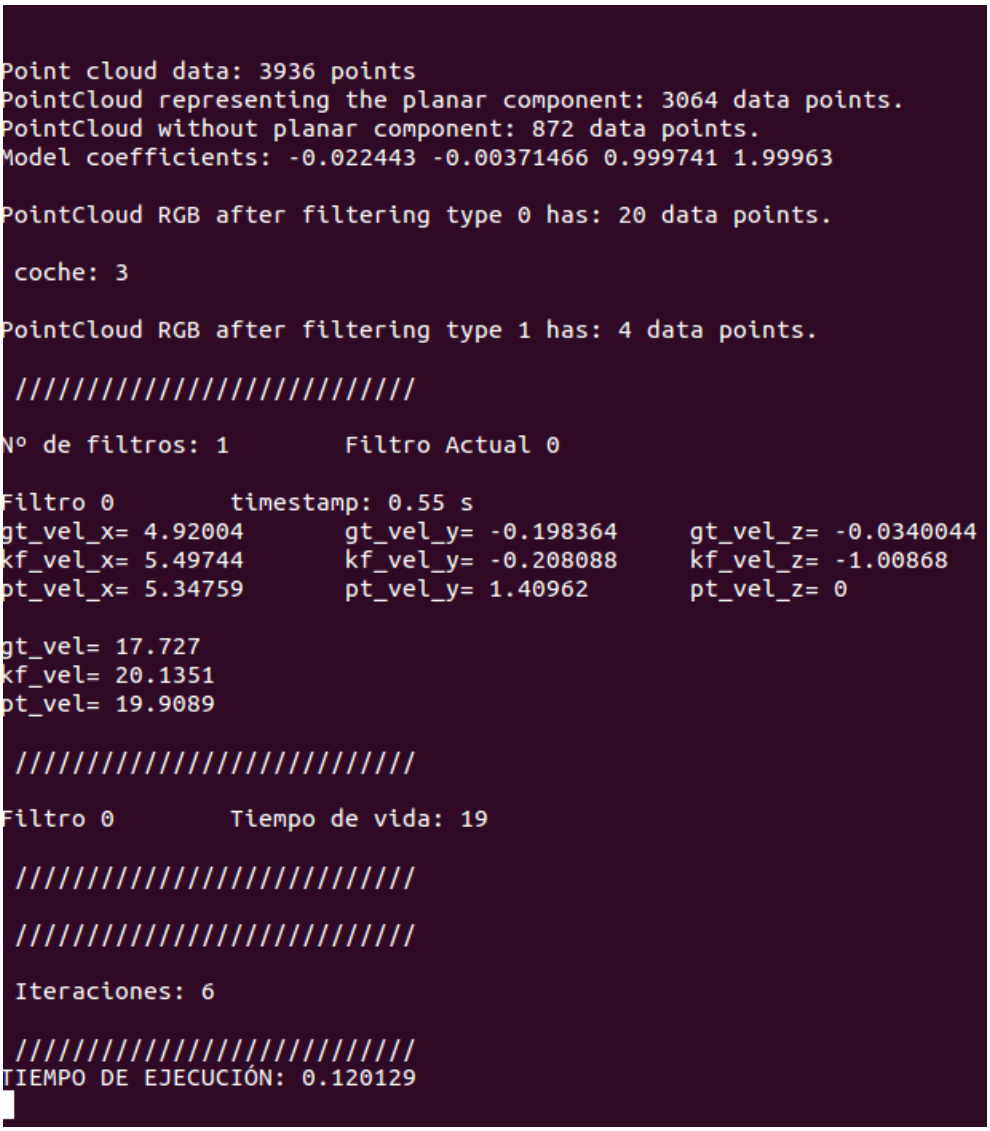

FIGURA MU.2-12: Información por pantalla del nodo principal.

BIBLIOGRAFÍA

[1] D. Held, J. Levinson, S. Thrun, S. Savarese, "Robust real-time tracking combining 3D shape, color, and motion", International Journal of Robotics Research, vol. 35, issue 1-3, págs. 30-49, 2016.

[2] Página Web Oficial Proyecto Smart Elderly Car. [Online]. Disponible: https://www.robesafe.uah.es/index.php/en/smartelderlycar

[3] Laura Leal-Taixé, Anton Milan, Konrad Schindler, Daniel Cremers, Ian Reid, Stefan Roth, "*Tracking the Trackers: An Analysis of the State of the Art in Multiple Object Tracking*", arXiv:1704.02781v1 [cs.CV], abril 2017.

[4] X. Jia, H. Lu, and M. Yang, "Visual tracking via adaptive structural local sparse appearance model", IEEE Conference on Computer Vision and Pattern Recognition, pp. 1822–1829, 2012.

[5] J. Kwon and K. M. Lee, "Visual tracking decomposition", IEEE Conference on Computer Vision and Pattern Recognition, pp. 1269–1276, 2010.

[6] M. Manz, T. Luettel, F. von Hundelshausen, H.-J. Wuensche, "Monocular model-based 3d vehicle tracking for autonomous vehicles in unstructured environment", IEEE International Conference on Robotics and Automation (ICRA), págs. 2465 – 2471, mayo 2011.

[7] A. Dewan, T. Caselitz, G.D. Tipaldi, W. Burgard, "Motion-based detection and tracking in 3D LiDAR Scans". IEEE International Conference on Robotics and Automation (ICRA), Estocolmo, Suecia, págs. 4508–4513, mayo 2016.

[8] Adam Feldman, Maria Hybinette, and Tucker Balch. "The multi-ICP tracker: An online algorithm for tracking multiple interacting targets", Journal of Field Robotics, 29(2):258–276, 2012.

[9] Q. Li, B. Dai, H. Fu, "LIDAR-based dynamic environment modeling and tracking using particles-based occupancy grid". Proceeding of the IEEE International Conference on Mechatronics and Automation, Harbin, China, págs. 238–243, agosto 2016.

[10] A. Harrison, P. Newman, "Image and sparse laser fusion for dense scene reconstruction" Proc. of the Int. Conf. on Field and Service Robotics (FSR), Cambridge, Massachusetts, julio 2009.

[11] M. Allodi, A. Broggi, D. Giaquinto, M. Patander, A. Prioletti, "Machine learning in tracking associations with stereo vision and lidar observations for an autonomous vehicle" Proceedings of the IEEE Intelligent Vehicles Symposium, Gothenburg, Suecia, págs. 648–653, junio 2016.

[12] A. Asvadi, P. Girao, P. Peixoto, U. Nunes, "3D object tracking using RGB and LIDAR data", Proceedings of IEEE International Conference on Intelligent Transportation Systems (ITSC), págs. 1255–1260, Rio de Janeiro, Brasil, noviembre 2016.

[13] Página Web Oficial de ROS. [Online]. Disponible: http://www.ros.org/

[14] E. Rohmer, S.P.N. Singh, M. Freese, "V-REP: A Versatile and Scalable Robot Simulation Framework", IEEE/RSJ International Conference on Intelligent Robots and Systems, noviembre 2013.

[15] R.B. Rusu, S. Cousins, "3D is here: Point Cloud Library (PCL)", IEEE International Conference on Robotics and Automation, 2011.

[16] Página Web Velodyne VLP-16. [Online]. Disponible: http://velodynelidar.com/vlp-16.html

[17] Página Web cámara estéreo Bumblebee® XB3. [Online]. Disponible: https://www.ptgrey.com/bumblebee-xb3-1394b-stereo-vision-camera-systems-2

[18] Página Web GPS HIPer Pro de Topcon. [Online]. Disponible: http://www.topcon.com.sg/survey/hiperpro.html

[19] Paquete de ROS but_velodyne para fusión sensorial. [Online]. Disponible: http://wiki.ros.org/but_velodyne

[20] M. Velas, M. Spanel, Z. Materna, A. Herout, "Calibration of RGB Camera with Velodyne LiDAR", WSCG, 2014.

[21] Wiki de ROS para filtros de mensaje. [Online]. Disponible: http://wiki.ros.org/message_filters

[22] C.Guindel, J. Beltrán, D. Martín, F. García, “Automatic Extrinsic Calibration for Lidar-Stereo Vehicle Sensor Setups”. IEEE International Conference on Intelligent Transportation Systems (ITSC), págs. 674–679, 2017.

[23] E. Romera, J.M. Álvarez, L.M. Bergasa, R. Arroyo, “ERFNet: Efficient Residual Factorized ConvNet for Real-Time Semantic Segmentation”. IEEE Transactions on Intelligent Transportation Systems, volumen 19, págs. 263-272, 2018.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá